**Setting up**

Go to terminal and desired folder: flutter create hostelfoods  
(creating the new flutter project)

cd hostelfoods

code . (to open the project in VS code)

ctrl+shift+p (to select the device: the emulator)

ctrl + f5 run without debugging

**Key Words in DART**

show, if, else, const, new, null, false, return, final

**Data Types in DART**

int, arrays, strings, double, bool

**Constant in DART**

Value of who’s value can’t be changed.

* Value of 10
* Your name

**02 types: Compile time vs Runtime**

Compile - process of packaging things up  
Run time - moment from running the program

#. Constants won’t change during Run time and compile time

**Variable in DART**

Variable is anything thing that contains a value that is not a constant.

* The data that it hold doesn’t need to be constantly same

var name = ‘Alex’

name = ‘Anthony’

final ( is a variable who’s value cannot change like a constant)

**Functions in DART**

void main() {

Void - Return value  
Name – main  
() where you write the functions arguments/parameters

Eg: String getFullName(String firstName, String lastName) {

return firtstName + ‘ ‘ + lastName; / return ‘$firstName’ ‘$lastName’;

}

**String - data type; function name – getFullName;**

**Parameter - firtstName, lastName**

Eg: void printMyName () {  
}

Eg: String printMyName () {

return ‘ ’ ;  
}

Eg: String getFullName(String firstName, String lastName) =>

‘$firstName’ ‘$lastName’;

**HOT RELOAD (ctrl + s)**

* It looks for changes we made to the code and execute only those changes.

**IF & ELSE**

void test() {

*const* name = 'Foo';

  if (name == 'Foo') {

    print('Yes, this is foo');

  } else if (name != 'Bar') {

    print('No this value is not Bar');

  } else {

    print("I don't know what this is");

  }

}

- as soon it goes to 1st branch, it doesn’t go to any other branches unless it doesn’t meet 1st condition

**OPERATOR**

* **Prefix operators, Infix operators, suffix operators**

= - is an assignment operator

== - is a conditional operator

+ -

- -

!= - not operator

/ - division

\* - multiplication

Infix operators🡪 when they have 2 values, one to each side of the operator ( / , \* )

*final* age = 20;

*final* halfOfAge = age / 20;

*final* doubleOfAge = age \* 20;

   print(halfOfAge);

   print(doubleOfAge);

Prefix operators 🡪 comes before what ever it has to it’s work on. (--)

  var age = 20;

*final* ageMinusOne = --age;

  print(age);

  print(ageMinusOne);

- operators can also be used on strings in DART.

*final* name = 'Foo Bar Bald ';

*final* nameTimes20 = name \* 100;

  print(nameTimes20);

- we can also create our own custom operators

**Lists in DART**

- index of list are starts from 0.

*final* names = ['Foo', 'Bar', ' Baz'];

*final* foo = names[2];

  print(foo);

out put -> Baz

index of 'Foo', 'Bar', ' Baz' – are 0,1,2

Another eg:

*final* names = ['Foo', 'Bar', ' Baz', 'Pog', 'Jar', ' Gaz'];

*final* length = names.length;

  print(length);

out put -> 6

names.length - dot in

var names = ['Foo', 'Bar', ' Baz', 'Pog', 'Jar', ' Gaz'];

  print(names.length);

  names.add('My Name');

  print(names.length);

out put -> 6  
 7

* Lists and array are almost same

**Sets in DART**

* List of unique “things”
* Data can’t be duplicated

  var names = {'foo', 'bar', 'bag'};

  names.add('foo');

  names.add('foo');

  names.add('goo');

  print(names);

out put -> {foo, bar, bag, goo}

hash codes - ???

**[Maps](#Maps_Dart) in DART**

* Maps are used to hold key-value pairs of information

*const* person = {

    'age': 20,

    'name': 'Foo',

  };

Out put -> {age: 20, name: Foo}

Keys – age, name   
Valus – 20, Foo

Can’t use used key again. Because keys need to be unique

But we can modify them,

  var person = {

    'age': 20,

    'name': 'Foo',

  };

  print(person);

  person['name'] = 'FOOOO';

  print(person);

out put -> {age: 20, name: Foo}

{age: 20, name: FOOOO}

But adding aa new key would be like,

 var person = {

    'age': 20,

    'name': 'Foo',

  };

  print(person);

  person['lastname'] = 'Bar';

  print(person);

out put -> {age: 20, name: Foo, lastname: Bar}

**Null safety – Sound null safety in DART**

**NULL VALUES -** Not a value, but the absence of a value.

What are null values? (Read the documentation)

**Making any type nullable**

* Use the question mark after the data type such as “String?”

Eg**:**

  String? name = null;

  print(name);

name = 'Foo';

  print(name);

output -> null  
 Foo

**Eg2:**

int? age = 20;  
   age = null;

**Eg3:**

int? age = 20;

    if (age == null) {}

**Eg4:**

List<String?>? names = ['Foo', 'Bar', null];  
  names = null;

* Both the container and the value it contains can be optional. (container – names)
* Can call it Optional list of optional strings or nullable list of nullable Strings

**Cherry-picking non-null values**

* Use the ?? operator

*const* String? firstName = null;

*const* String? middleName = 'Bar';

*const* String? lastName = 'Baz';

  if (firstName != null) {

    print("first name is the first non-null value");

  } else if (middleName != null) {

    print("middle name is the first non-null value");

  } else if (lastName != null) {

    print("last name is the first non-null value");

  }

**Other way – with ?? OPERATOR**

void test(String? firstName, String? middleName, String? lastName) {

*final* firstNonNullValue = firstName ?? middleName ?? lastName;

  print(firstNonNullValue);

}

test('Goo', null, 'Baz');

output -> Goo

**Null-aware assignment operator**

* Use the ??= operator

void test(String? firstName, String? middleName, String? lastName) {

String? name = firstName;

name ??= middleName;

name ??= lastName;

print(name);

}

test(null, null, 'Baz');

output -> Baz

**Conditional Invocation**

* Use the ?. syntax to conditionally invoke a method or property

**Old way**

void test(List<String>? names) {

  if (names != null) {

*final* length = names.length;

  }

or

void test(List<String>? names) {

*final* int length;

  if (names != null) {

    length = names.length;

  } else {

    length = 0;

  }

}

**Better way – with conditional invocation**

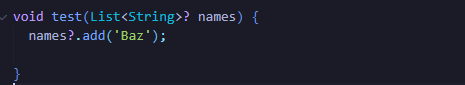
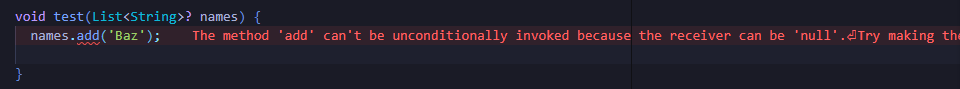
void test(List<String>? names) {

*final*  length = names?.length ?? 0;

  length;

}

**Can do it for functions as well.**



Without ?. it shows an error

**Null aware operations,  
?? - Decision making infix operator  
?. - Invocation operator  
??= - Assignment operator**

* **Read Documentation about null safety**

**Chapter 06**

**Dart enumerations, classes and objects**

**Enumerations**

* **Named list of related items**

enum PersonProperties { firstName, lastName, age }

void test() {

  print(PersonProperties.firstName.name);

}

**Output -> firstName**

**Switch Statement**

* To avoid if else over enumerations

- Normal way of typing if statements

enum AnimalType { cat, dog, bunny }

void test(AnimalType animalType) {

  if (animalType == AnimalType.cat) {

    print("I love cats");

  } else if (animalType == AnimalType.dog) {

    print("I hate dogss");

  } else if (animalType == AnimalType.bunny) {

    print("I am fine with rabbits");

  }

}

Recommended way is using switch statements

enum AnimalType { cat, dog, bunny }

void test(AnimalType animalType) {

  switch (animalType) {

    case AnimalType.bunny:



      print("I am fine with rabbits");

      break;



    case AnimalType.dog:

      print("I hate dogss");

      break;

    case AnimalType.cat:

      print("I love cats");

  }

  print("Function is finished");

}

Out put -> I love cats  
 Function is finished

Also, in these case we can use a typical if statement

enum AnimalType { cat, dog, bunny }

void makeSureThisIsACat(AnimalType animalType) {

  if (animalType != AnimalType.cat) ;

**Classes**

* Grouping of various functionalities into one packageable piece of data
* Instances are objects, objects are created from classes
* Every class can be instantiated

void main() {

  runApp(*const* MyApp());

}

class Person {

  void breathe() {

    print('Breathing');

  }

  void run() {

    print('Running');

  }

}

void test() {

*final* person = Person();

  person.run();

  person.breathe();

}

Output -> Running  
 Breathing

* Instance member - Some functionality that is only available at the instance level

**Objects**

* An object is an instance of classes

void test() {

  Person(); This is an instance (a useless object

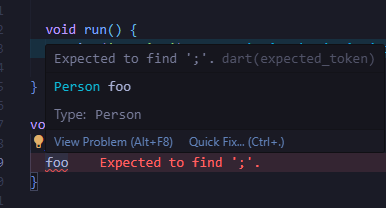


}

void test() {

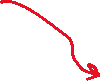
*final* foo = Person(); Now the foo is an object



}

When write it’s name,

It will tell it’s a person



**Constructors**

* A special logic in a class that constructs or initializes or builds that class’s instance
* Allow you to create an instance of a class with optional parameters

class Person {

*final* String name;

*// Constructor using shorthand syntax*

  Person(this.name);

}

void test() {

*// Creating an instance of the Person class*

*final* foo = Person('Foo Bar');

*// Accessing the 'name' property of the created instance*

  print(foo.name);

}

**Methods**

* Method of a class is a function on the class (usually referred to as an instance method)

class Person {



*final* String name; name is an instance variable

  Person(this.name);

}

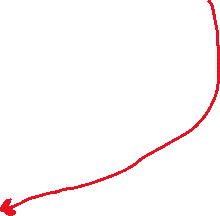


void test() { this is a function

*final* foo = Person('Foo Bar');

  print(foo.name);

}



class Person {

*final* String name;

  Person(this.name);

  void printName() {

    print(this.name); “this” key word basically refers to the

  } current instance of this class



}

void test() {

*final* foo = Person('Foo Bar');

  print(foo.name);

}

* It’s usually better not prefix your instance functions inside the class itself using “this” keyword

class Person {

*final* String name;

  Person(this.name);

  void printName() {

    print('I will now print he name of this person');

    print(name);

  }

}

void test() {

*final* foo = Person('Foo Baz');

  foo.printName();

}

Out put -> I will now print the name of this person  
   Foo Baz

**Inheritance and subclassing**

* Classes can do a lot for us including inheritance and allowing us to subclass them
* It allows you to define a class and then to add more functionalities to that class into a new class

class LivingThing {

  void breath() {

    print('Living thins is breathing');

  }

  void move() {

    print('I am moving');

  }

}



class Cat extends LivingThing {}

This keyword is used to mean sub class

void test() { or inheritance

*final* fluffers = Cat();

  fluffers.move();

  fluffers.breath();

}

Out put -> I am moving  
 Living thing is breathing

**Abstract Classes**

* They can contain logic too!



abstract class LivingThing {



  void breath() {

    print('Living thins is breathing'); makes no difference

  }

  void move() {

    print('I am moving');

  }

}

class Cat extends LivingThing {}

void test() {

*final* fluffers = Cat();

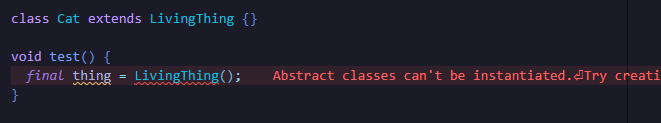
  fluffers.move();

  fluffers.breath();

}

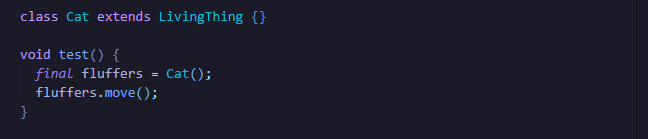
Out put -> I am moving  
 Living thing is breathing

* An abstract class is just like a normal class that can’t have instances





* An abstract class is a class that groups logic into itself with the sole purpose of other classes using its functionalities
* This is class that has some utilities that has some code that is supposed to be used in other normal classes.



` Now it’s correct



* It’s a class that cannot be instantiated, it’s usually like a utility class that other classes are supposed to inherit from

**Factory Constructors**

* Can return instances that are not of the same class!



This is a constructor



Keep the cursor on variable (name) and press Ctrl + . (in VS code) to create a constructor

class Cat {

*final* String name;

  Cat(this.name); This is a normal constructor

}



void test() {

*final* fluffBall = Cat('Fluff Ball');

  print(fluffBall.name);

}

* A factory constructor is a way for you to construct instances of your class using convenience functions



class Cat { Any class called Cat which is this Class

*final* String name;

  Cat(this.name); Has a function called fluffBall



  factory Cat.fluffBall() {



    this.name = 'Fluff Ball'; This is how you basically create an



  } instance of our class

}

* I have a constructor called fluffBall whose return value is a Cat but inside I have some special logic that packages it up, just like you do it in an actual factory

class Cat {

*final* String name; This is how you create a

  Cat(this.name); factory constructor

  factory Cat.fluffBall() {



    return Cat('Fluff Ball 2');

  }

}

void test() {

*final* fluffBall = Cat.fluffBall();

  print(fluffBall.name);

}

Out put-> Fluff Ball 2

* Factory constructor is really important when you are talk about class clusters

**Customer Operators**

* You can define custom operators on your own classes in dart
* A custom operator in dart is an operator that allows us in dart to override the ability of your class to be compared to or add to or subtracted from or multiply with something with your own logic.

class Cat {

*final* String name;

  Cat(this.name);

} Output-> 'They are not equal'

void test() {

*final* cat1 = Cat('Foo');

*final* cat2 = Cat('Foo');

  if (cat1 == cat2) {

    print('They are equal');

  } else {

    print('They are not equal');

  }

}

Additional Research about the above code from CHATGPT

In Dart, the **==** operator checks for object identity by default, not for structural equality. When you create two instances of a class, even if the instances have the same field values, they are considered different objects in terms of identity unless you explicitly override the **==** operator to compare their contents.

In your case, the **Cat** class does not **override** the **==** operator, so it uses the default behavior, which checks for object identity.

How you can override the == operator in your Cat class to compare the content of the objects:

class Cat extends Object {

*final* String name;

  Cat(this.name);

  @override

  bool operator ==(covariant Cat other) => other.name == name;

  @override

  int get hashCode => name.hashCode;

}

void test() {

*final* cat1 = Cat('Foo');

*final* cat2 = Cat('Foo');

  if (cat1 == cat2) {

    print('They are equal');

  } else {

    print('They are not equal');

  }

}

OUTPUT -> They are equal

* Override is a very important term, whenever you have a class and you are inheriting from that class (Class Cat)
* By default, in DART every class inherits implicitly from the object class

class Cat extends Object {



*final* String name;



  Cat(this.name);

  @override

class Cat {

*final* String name; This is exact same as omitting writhing

  Cat(this.name); the above two words



  @override

* Cat comes from object
* An object internally already defines an operator called == that returns a Boolean.
* A Boolean is a value that can either be true or false.
* bool is a Data Type
* operator is a keyword in DART

  @override

  bool operator ==(covariant Cat other) => other.name == name;



* That operator tells Dart that it going to change (as in override) the functionality of this == operator that is defined at the object level with our own implementation
* Since an operator in itself is function, it has a ()
* In the (), you get value the that, that operator is comparing your class with
* That value that your class is being compared to also need to be a Cat
* covariant is a keyword, It tells DART although at the object level we said that the parameter comes in is of type object but in our case we are sure that the value that comes in this function is actually a Cat.
* hashCode: Is a special identifier that you assign to your instance of classes that is then used inside collections.
* If you put instances of your Cat class inside a [Map](#Maps_Dar) then the way DART knows that the keys inside it’s Maps are actually unique, it’s using this hash value.



@override

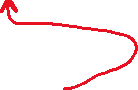
  bool operator ==(covariant Cat other) => other.name == name;

  @override

  int get hashCode => name.hashCode;



* Override the value of hashCode int get



* Create method hashCode. [(Ctrl+.)](#method)
* We change the super to name (Don’t get super’s hashCode, get name’s hashCode)
* We are creating an identifier for our Cat class. Which allows DART to understand that if this Cat classes instance are placed inside collections such as a Map or a Set then it will know if they are unique or not.

class Cat extends Object {

*final* String name;

  Cat(this.name);

  @override

  bool operator ==(covariant Cat other) => other.name == name;

  @override

  int get hashCode => name.hashCode;

}

void test() {

*final* cat1 = Cat('Foo');

*final* cat2 = Cat('Foo');

  if (cat1 == cat2) {

    print('They are equal');

  } else {

    print('They are not equal');

  }

}

OUTPUT -> They are equal



**End of Chapter 6 - Reading the Official documentation will help.**

**Next - future, async operations, streams, generators**