OOP : object oriented programming :-

← بتساعد بالي تنظيم الكود أكتر ميكونش كله مكتوب في ملف واحد فقط

← الكود كله بيتقسم لحاجتين [Classes - objects]

← الـ Class هو قالب "Blueprint" يعني شرح و وصف للحاجة مش الحاجة نفسها [الرسم الهندسي للشئ]

← الـ object هو الحاجة نفسها اللي بيتم إنشائها عن طريق الوصف في الـ class نفسه [التصميم الحقيقي بناءً على الرسم الهندسي]

* الـ OOP بتجعل الكود منظم وأسهل في الفهم - بيكون الكود مش معقد وعملية الـ debugging بتكون أسهل بكتير لإنك تكدر الخطأ أسهل إيجاده وبتقلل عملية الـ Repetition اللي في الكود عشان لو عندي شخصين ليهم نفس الـ properties مكتبش لكل واحد فيهم كود مختلف

* أهم الحاجات في الـ (OOP)

Class - object - Encapsulation - Inheritance
Polymorphism - Abstraction

1] Class

الكلاس بيتكون من حاجتين أساسيتين [الشئ ده خواصه إيه و بيعرف يعمل إيه]
Attributes / methods

```
class Users {
    String? name;
    String? email;
    int calcSalary() {};
}
```

كل كلاس له خواص بتحدده و حاجات بيقدر يعملها

Subject | Date

2 object

ده النسخة اللى بيتم أخذها من ال class ولها نفس الخصائص ونقدر نعمل كل ال Methods

Users us1 = Users();

هنا من خلال us1 نقدر نديله name و email والخ ونعمل كل ال methods جوا الكلاس

us1.name = "Mahmoud"

int x = us1.Calc_Salary

أقدر آخذ عدد لا نهائى من النسخ من ال class الرئيسى وكلهم مختلف فى الأماكن والقيم بتاعتهم

⊗ Constructor

عبارة عن method خاصة بتشتغل بشكل تلقائى لما أجى أخد object من الكلاس دة اللى بيبنى ال object ويديله القيم بتاعته الإبتدائية مثلاً لأنه بيتم إستدعائه تلقائياً قبل أى حاجة أول ما تاخد object من الكلاس

● Default Constructor ← بيكون معمول تلقائى وفاضى مفهوش حاجة دة بيحصل لو أنا مطلعتش أى Constructor فى الكود غير كدة هو موجود

ال Constructor لازم يحمل إسم ال class / ملوش return type ولا حتى void

```
Users (String name2) {
    name = name2;
}
```

⇒ Users(this.name) كلمة this بتشاور على المتغير اللى موجود فى الكلاس نفسه ⇐ معناها ياخد الإسم اللى اليوزر دخله ويحطه فيه

```
Users us2 = Users("mohamed");
⇒ us2.name → mohamed
```

Subject Date

⇒ Optional Parameters in Constractor

Users ([this.name]); ⇒ optional Param → functions

سواء عرفت له قيمة او لا البرنامج هيشتغل عادي

⇒ named Parameters

Users ({this.name}); | → required named
users({required this.name});

نفس كل اللي في ال parameters اللي في ال functions

(*) named Constractors

ايه الغرض منه انه يكون عندنا كذا Constractor جوا ال class

لو عشان يكون عندنا كذا طريقة مختلفة للتهيئة بناء على البيانات اللي عندنا

1 Data mapping ← كانت البيانات جاية "JSON" وعاوز تحولها لـ object

2 Predefined States ← لو عاوز تعمل نسخة بيانات ثابتة ومعروفة سابقاً

* Defaut

User (this.name);

* Named

~~User (this.name)~~

User.admin (this.name): name = "="

User.guest (); name = 'Guest';
age = 18,
role = 'USER'

User.FromJson(Map<string, dynamic> json);
name = json['name'],
age = json['age'],
role = json['role'],

الإستدعاء → var user1 = User.admin("mahmoud");

Subject

Date

[3] Encapsulation (underscore befor the Variable)

"التغليف" بيحدد مين مسموح له يغير بيانات الـ object وعين لأ
بيدعم مفهوم الـ Data Hiding بيخفي ويمنع الوصول للـ State الداخلية
بيانات الـ object إلا بطرق محددة علشان تعمل Validation للداتا قبل ماتحطها

الـ Setters - getters ⇐ دول اللي بيسمحوا إننا نعمل
ونـ acsses قيمة المتغيرات بشكل آمن (Validation)

```
double _balance = 0;
double get blance => _balance;
set deposit(double amount){
    if(amount > 0){
        ~~~
    }else{// منفعش}
```

[4] Inheritance

الوراثة إنه لو كنا class ليهم خصائص كتير متشابهة بدل ما نعمل
الخصائص دي مرتين ونعيد كتابة الكود نعمل class أب ليهم يحمل
الخصائص المشتركة بينهم

(*) single Inheritance

Class Employee{ } ⇒ class Manager extends Employee{ }

ليه علاقة (Is-A) مباشرة ⇐ الـ Manager هو Employee أيضًا

(*) multi-level Inheritance

Person → Employee → manager

كل واحد بيورث من اللي فوقه

(*) Hierarchical Inheritance

Animal → Cat, → Dog

كذا كلاس بيورثوا من نفس الكلاس

(*) Inheritance of Constructor

لو عندك في الـ Parent ← Constructor بتمرر له Parameter هيظهر مشكلة في الـ Child
لازم تعمل Constructor في الإبن وتمرر له الـ Parameter عن طريق
كلمة super ←

super.name ← بتوصل للـ name إلي موجود في الـ Parent عكس كلمة this
اللي بتوصل للي في نفس الـ class مش الأب

* لو انت عملت object من الـ Child هيستدعي Constructor الـ Parent
أولاً ثم هيستدعي الـ Child بعده لازم

```
Users ({this.name});
Employee(name) : super(name: name) { ... }
```

```
Users.maneger( this.name) ;
Employee.maneger( name ) ; super .manger(name) { ... }
```

named

```
display () { print ("Parent") }
displayEmployee () { super.display();
                     print ("Employee"); }
```

function

(*) [5] Polymorphism

بتكون قدرة الـ object على الظهور بأكثر من صورة ⇐ (Method overriding)
لو عندك Shape → square / circle ← فيه دالة تحسب المساحة في الـ shape
لكن طريقة حساب المساحة في كل من الأبناء مختلفة [اسم واحد ووظيفة مختلفة]

① overriding ← بتعيد كتابة نفس الدالة بنفس كل حاجة ماعدا
إنك بتغير في محتوى الدالة نفسها

② overloading ← مفيش نفس الدالة بنفس الـ parameters ← هو هنا بيكون
بيحطوها (named - optional) parameters

Mohamad & Amira

⊛ Up Casting in Polymorphism

```
List<Animal> clinic = [Dog(), Cat(), Dog()];

for (var animal in clinic) {
    animal.makeVoice();
}
```

⇒ هنا يحصل الـ Polymorphism لحظة التشغيل حيث يتم استدعاء الدالة المناسبة