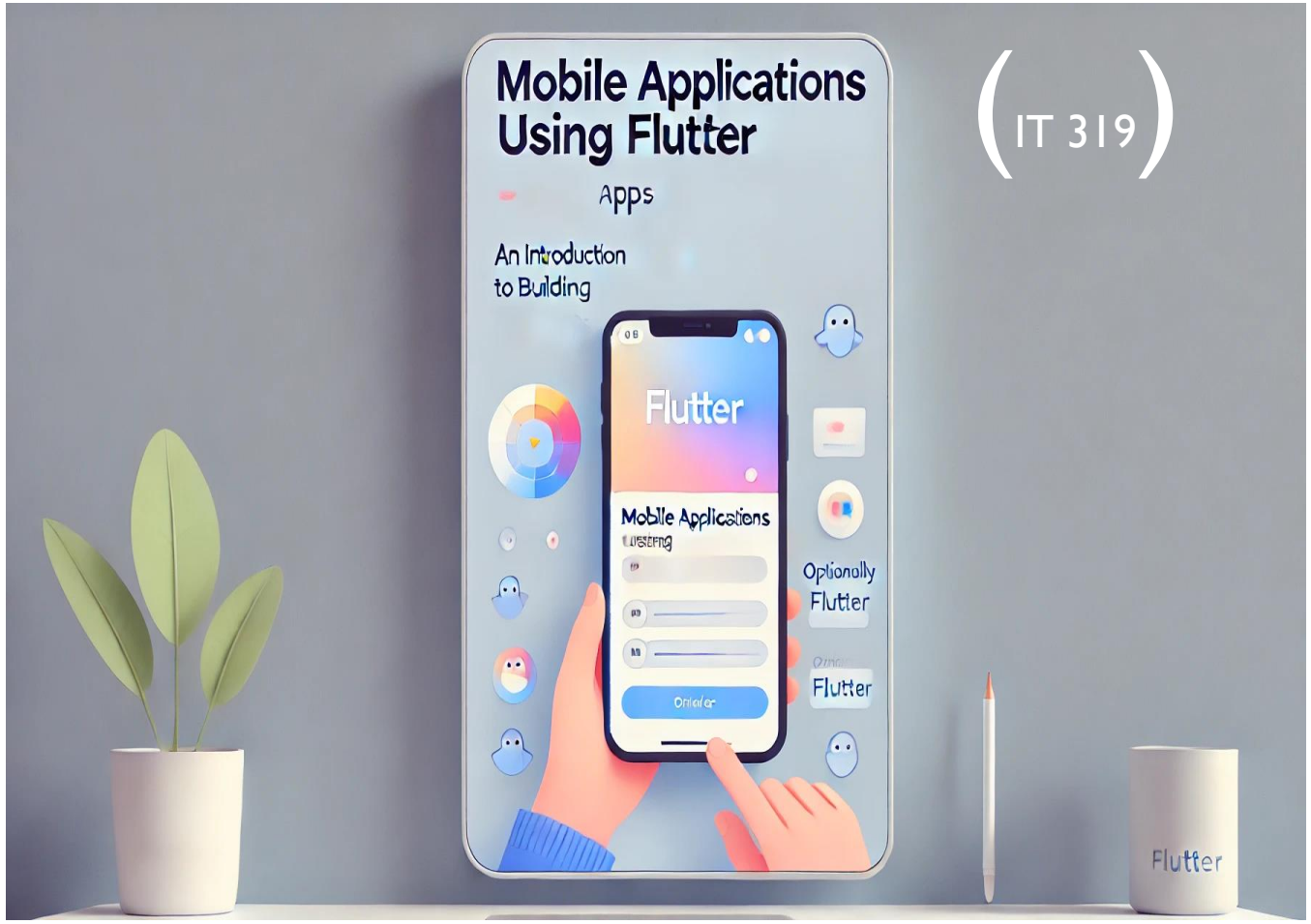
# MOBILE APPLICATIONS

OOP I earlier




(IT 319)

IT DEPT.

TIU

3RD GRADE

2025-2026 Fall Term

Lect. Mohammad Salim Al-Othman

Week 1-2

# CONTENTS

## Syllabus List

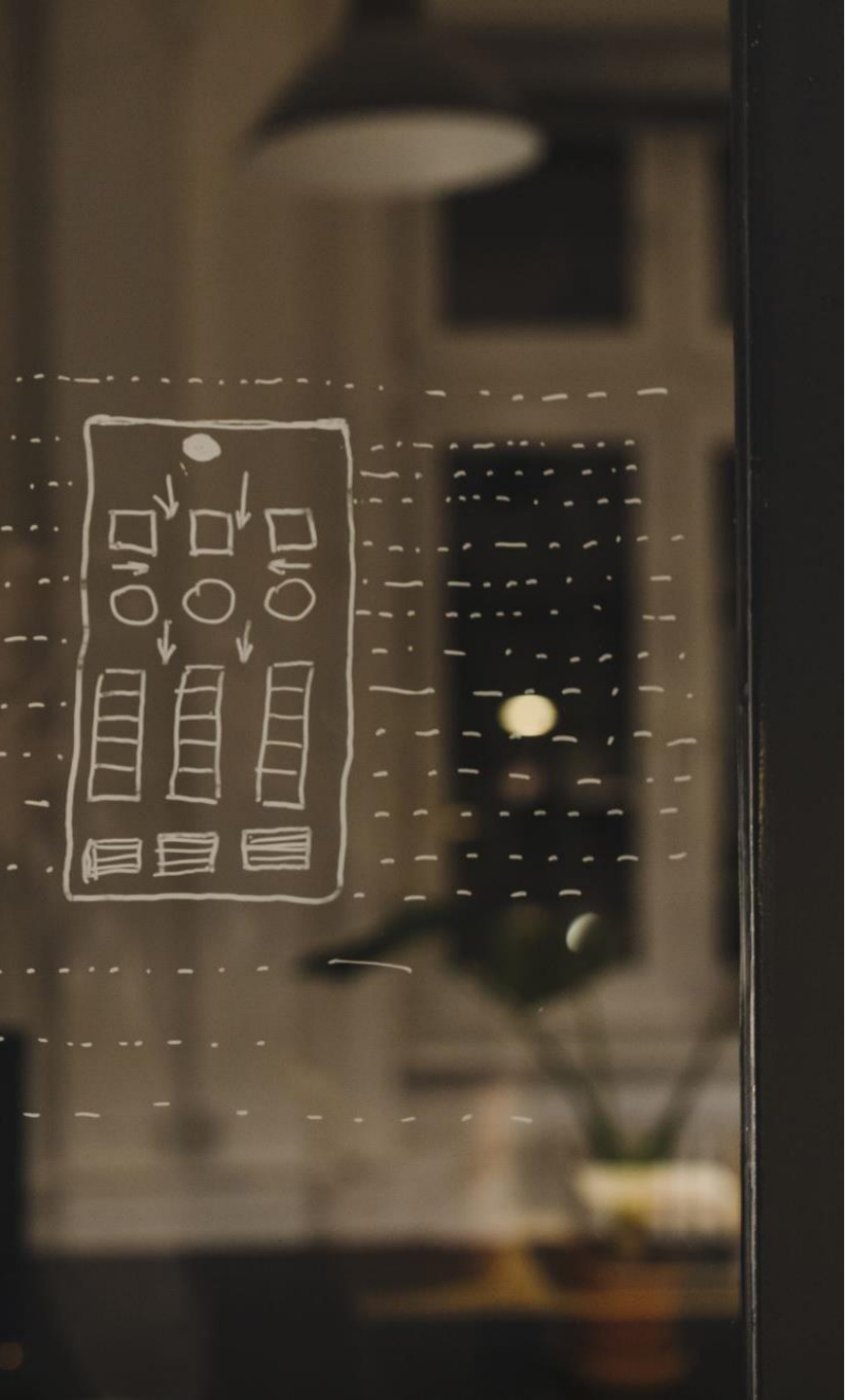
(Does not effect minor grades. The information is presented from syllabus)

| **Method** | **Quantity** | **Percentage (%)** Homework#2#5 |
|---|---|---|
| Quiz | 2 | 5 |
| Homework | 2 | 5 |
| Project | 1 | 15 |
| Midterm Exam | 1 | 20 |
| Laboratory | 1 | 5 |
| **Total** | | **60** |

# CLOS

COURSE/STUDENT LEARNING OUTCOMES:

1. Enhanced grasp of OOP principles.
2. Apply OOP concepts in Dart and Flutter to build Mobile Apps.
3. Implement OOP in Flutter Apps.
4. Foster teamwork by developing real-world projects.



# CONTENT

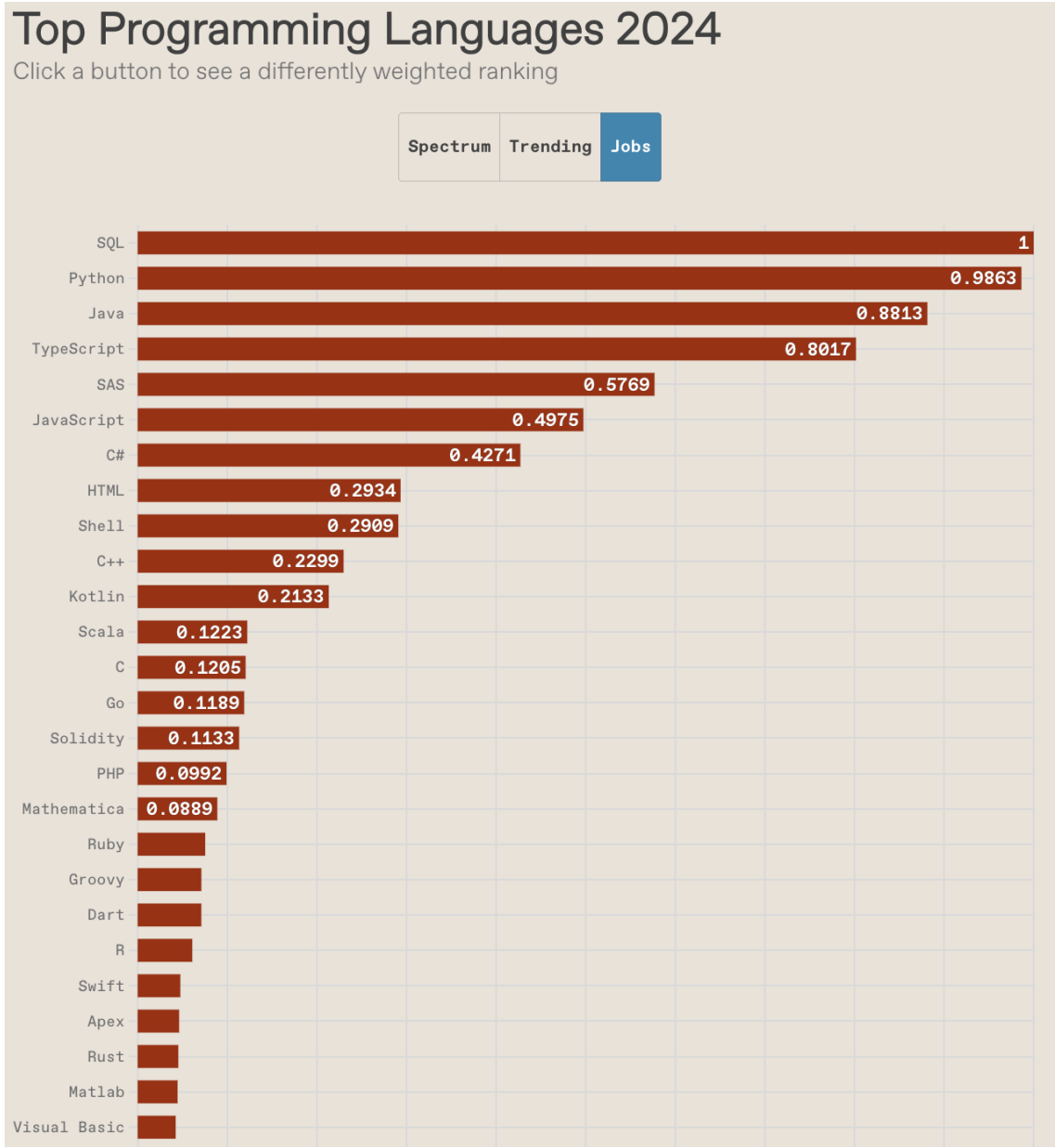
- Introduction to OOP , Class diagram with Dart Packages
- Build Your First Flutter App and Everything's a Widget, start to build a full-featured recipe app named Fooderlich
- Finish building the full-featured recipe app named Fooderlich and Understanding widgets and Stateless widgets and build our personal profile application (HW1)
- Application bar, list view and build a custom widget(HW2)
- Handle user input and Handle gestures and responsive design
- Material Design, Build for Android and iOS platforms, Colors and Themes

# Key facts about your topic

- Object-oriented programming is a programming paradigm based on the concept of "**objects**", which can contain **data**, in the form of **fields**, and **code**, in the form of **procedures**.
- A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated.
- In OOP, computer programs are designed by making them out of objects that interact with one another.
- OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types. Dart, Java, C++, and C# are OOP languages.

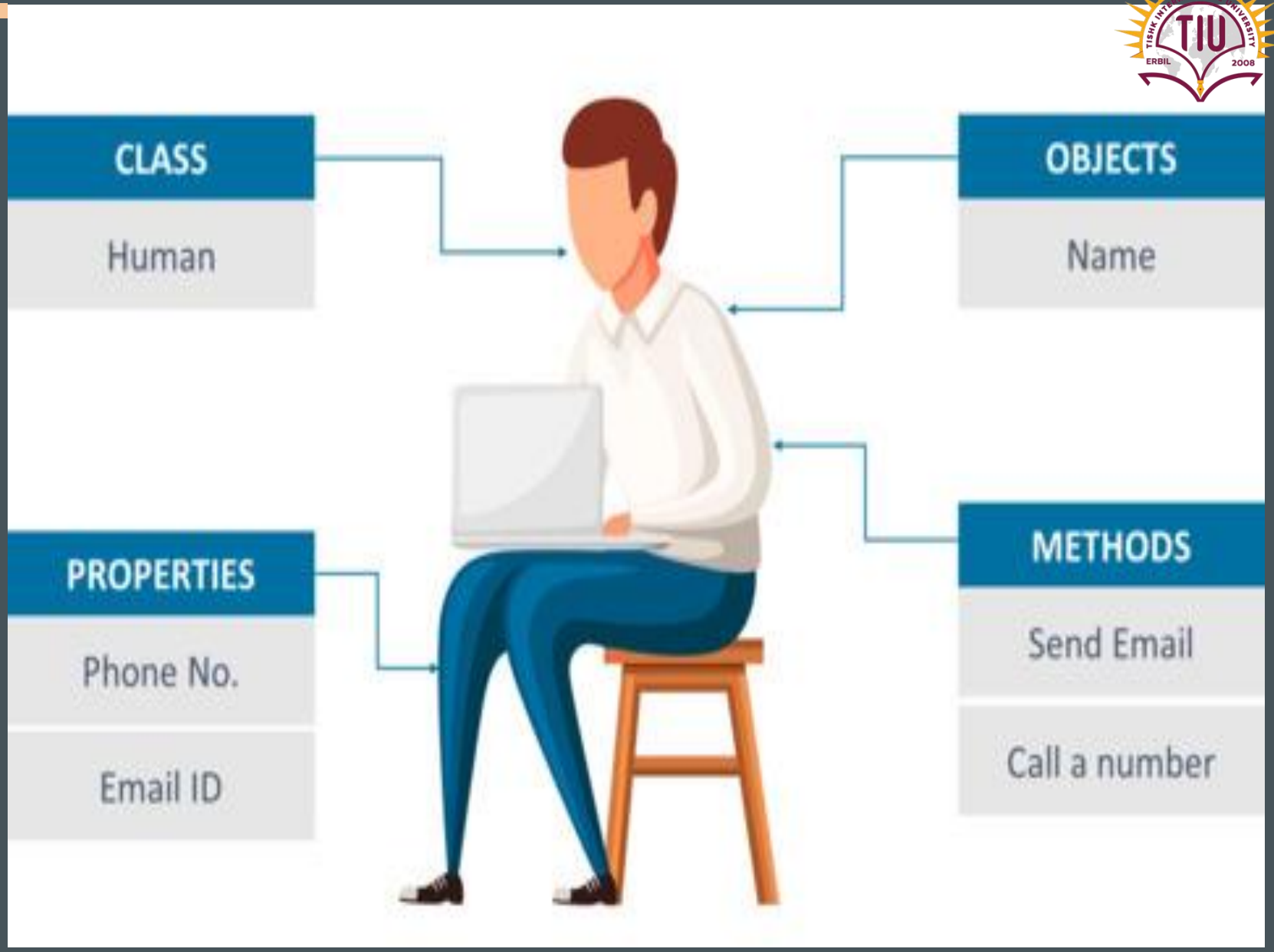
# TRENDS

- Rankings are created by weighting and combining metrics from eight sources:
- CareerBuilder, GitHub, Google, Hacker News, the IEEE, Reddit, Stack Overflow, and Twitter.



# OOP LANGUAGES

JAVA, C++, C#, PYTHON, R, PHP, VISUAL BASIC.NET, JAVASCRIPT, RUBY, PERL, SIMSCRIPT, OBJECT PASCAL, OBJECTIVE-C, DART, SWIFT, SCALA, KOTLIN, COMMON LISP, MATLAB, AND SMALLTALK.

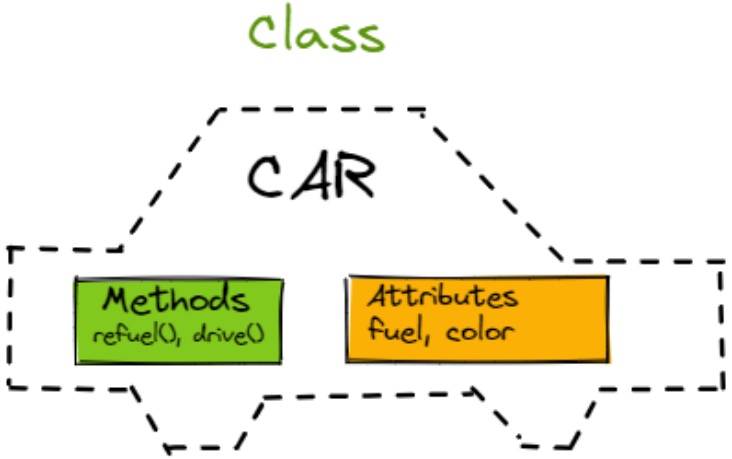


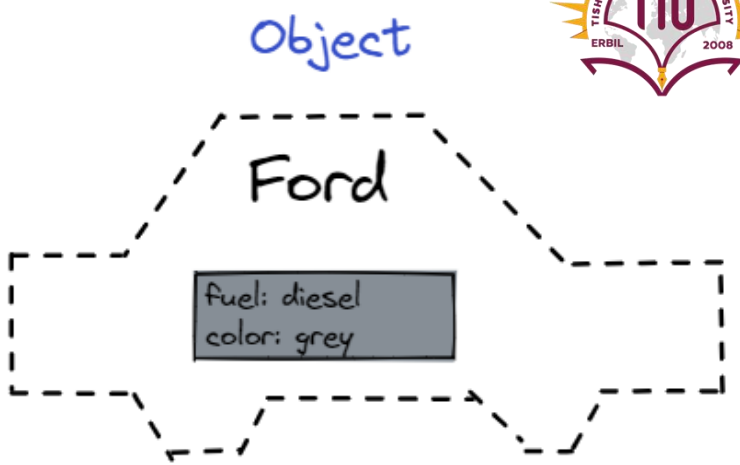
# Various OOP features can be implemented in Dart they are :

- Classes
- Objects
- Data Encapsulation
- Inheritance
- Polymorphism

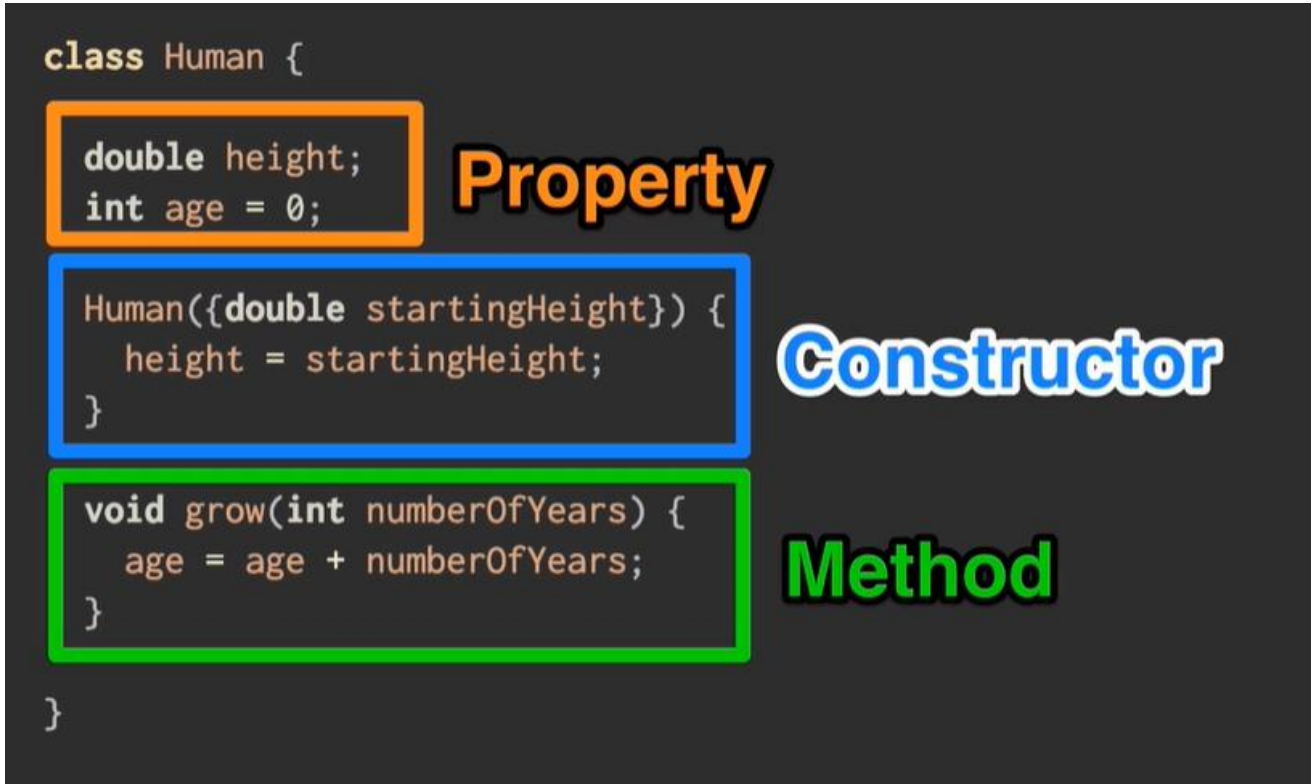
FEATURES

# 1.CLASSES



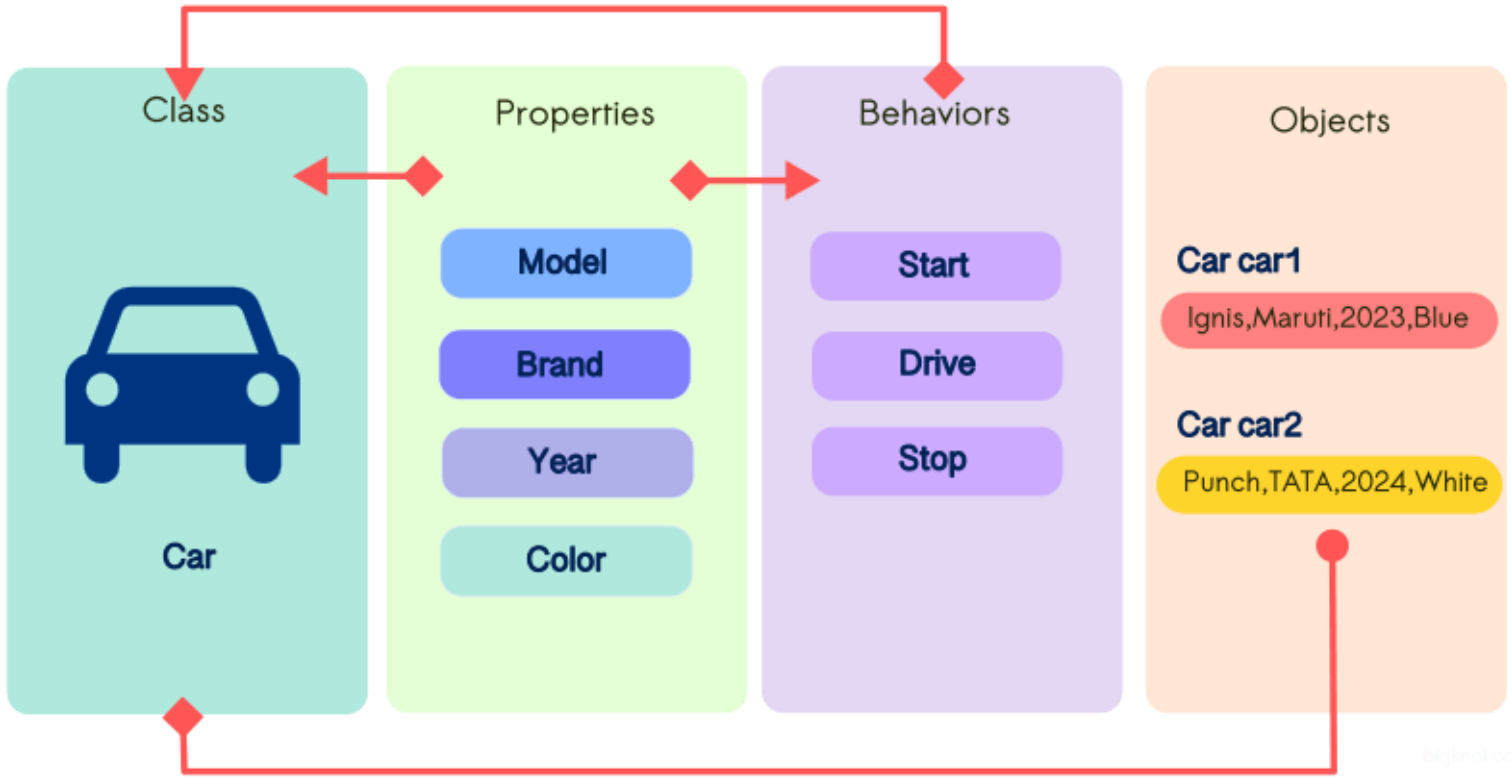




- Class is a user defined data type and it contains it's own data members(Constructors , getters and setters) and member functions.
- A class encapsulates data for the object. A class in Dart can be decalred by using the keyword class followed by the class name and the body of the class should be enclosed with a pair of curly braces {}
- One important thing to note is the rules of identifers must be followed while declaring a class name.
- A class contain constructors , fields , functions , setters and getters.
- Syntax for class declaration



# 2. OBJECTS

- **Objects** are concrete representations of classes. Each object can hold specific values for the attributes defined in its class.
- **State:** The current values of the object's properties (attributes). For example, an object of the class Car may have attributes like **color**, **model**, and **speed**.
- **Behavior:** The **methods** (**functions**) that define what an object can do. For instance, a Car object might have methods like **accelerate**() and **brake**().

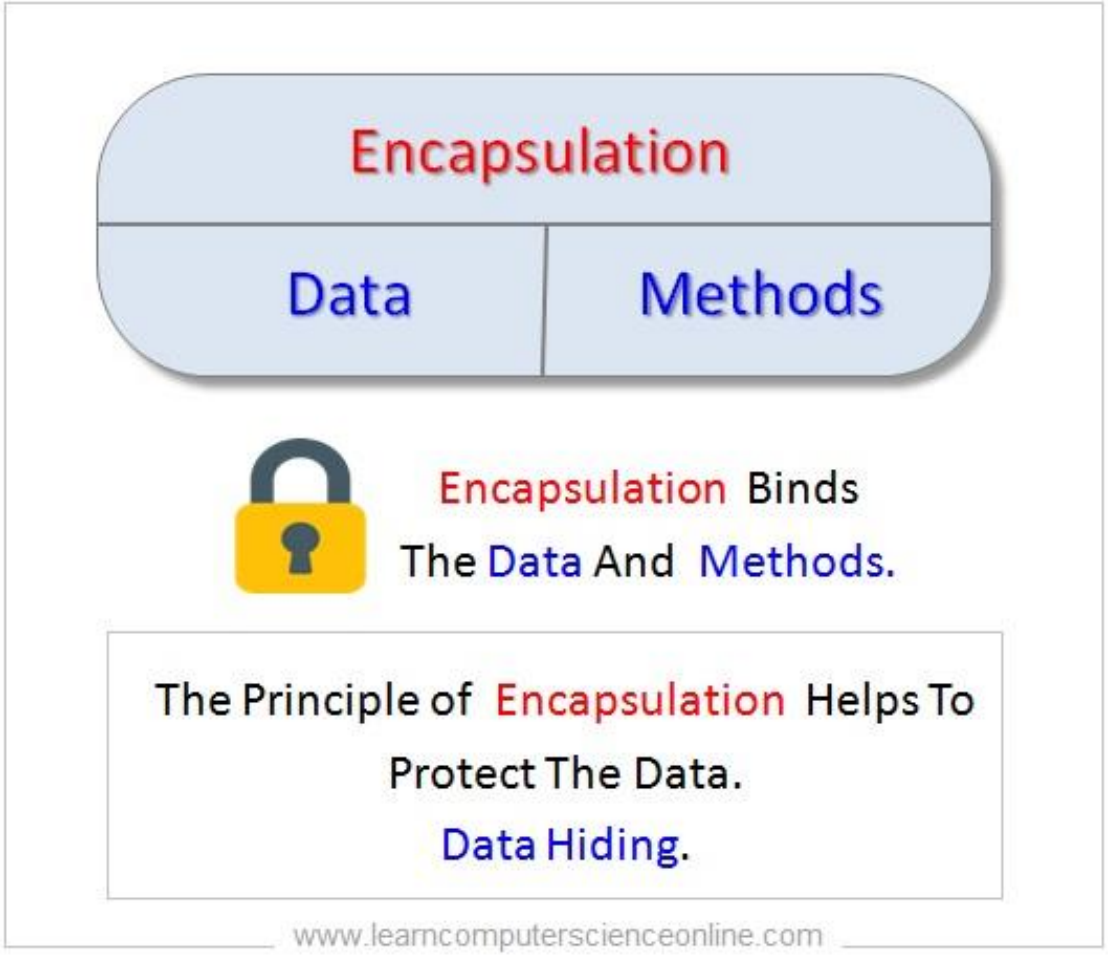





Objects are fundamental to OOP, allowing developers to create modular, reusable, and maintainable code. They facilitate a clear structure and organization by representing real-world entities and their interactions within a program.

# 3. DATA ENCAPSULATION

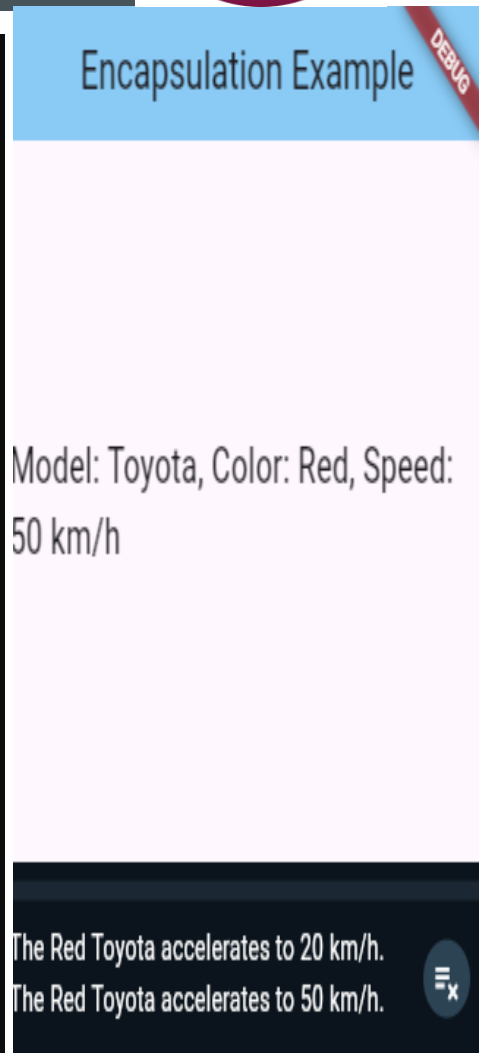
- Encapsulation in Flutter, as in other Object-Oriented Programming (OOP) languages, refers to the bundling of data (attributes) and methods (functions) that operate on that data within a class, while **restricting direct access** to some of the object's components. This helps in protecting the integrity of the data and promoting a modular design.
- Encapsulation in Dart happens at the **library level** and not at the **class level**.
- Any **identifer** that starts with an underscore _ is **private** to its **library**.



# 3. DATA ENCAPSULATION

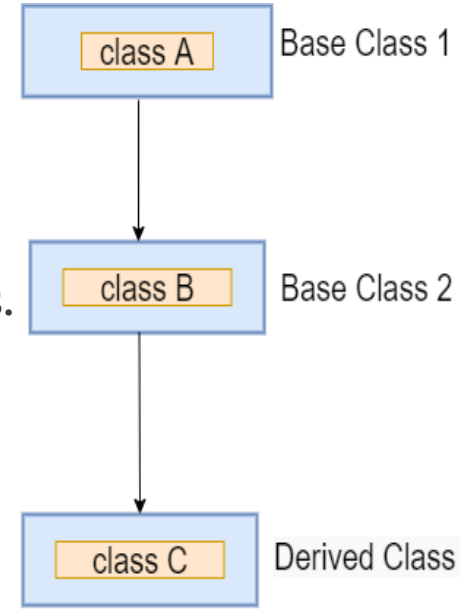
```
class Car {
  // Private attributes (encapsulation)
  String _color;
  String _model;
  int _speed;

  // Constructor
  Car(this._color, this._model) : _speed = 0;

  // Public method to accelerate the car
  void accelerate(int increment) {
    _speed += increment;
    print('The $_color $_model accelerates to $_speed km/h.');
  }

  // Public method to get the car's details
  String getDetails() {
    return 'Model: $_model, Color: $_color, Speed: $_speed km/h';
  }

  // Public method to get the current speed
  int get speed => _speed;
}
```

```
class CarInfo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Create an instance of the Car class
    Car myCar = Car('Red', 'Toyota');

    // Access the public methods
    myCar.accelerate(20);
    myCar.accelerate(30);

    return Text(
      myCar.getDetails(),
      style: TextStyle(fontSize: 20),
    );
  }
}
```



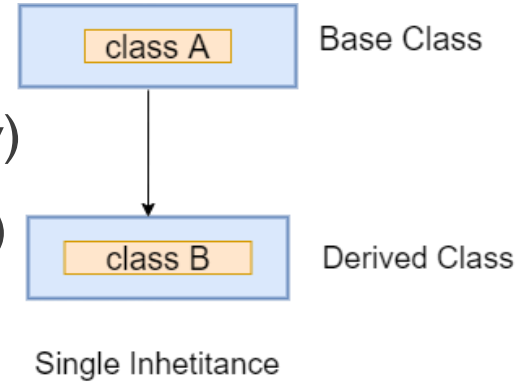
# 4. INHERITANCE


- Inheritance means the ability to create new classes from an existing one.
- The new created classes are called sub classes or child classes.
- The class from which sub classes are derived is called the super class or a parent class. A class is inherited from another class by using the extend keyword.



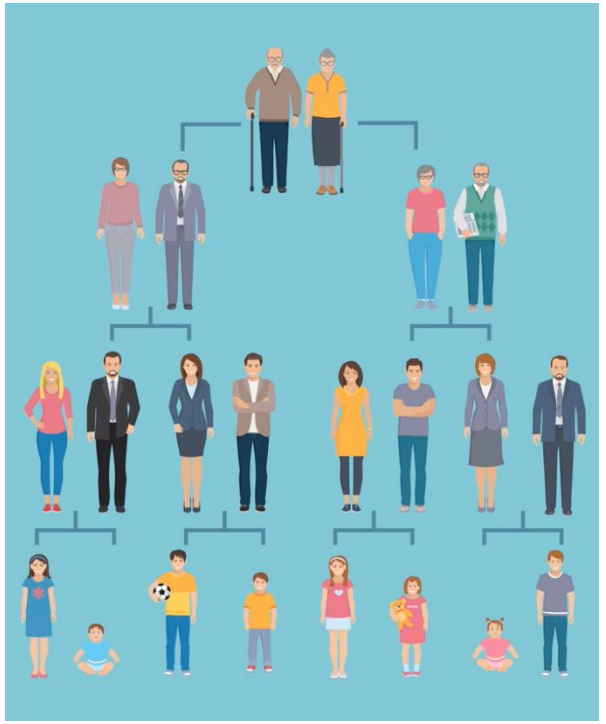

Dart supports the following types of Inheritance :

- Single (one child class is inherited by one parent class only)
- Multi level (child class can inherit from another child class)
- Dart does not support Multiple Inheritance.




- The **super keyword** is used to refer to **immediate parent of a class**. The keyword can be used to refer to the **super class** version of a **method** or a **variable**.

# INHERITANCE IN ACTION

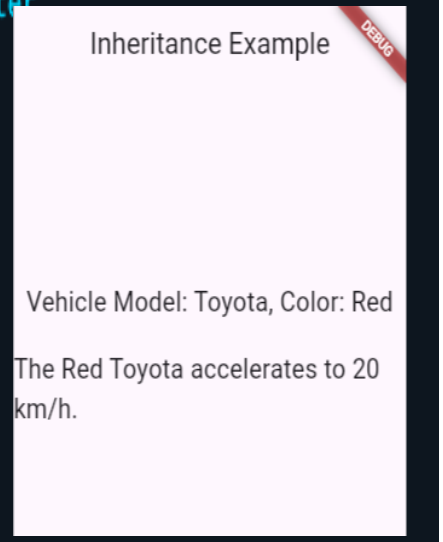


- This principle leads us to an important concept in object-oriented programming that allows a class to inherit properties and methods from another class and to extend them. So let's define:
- **sub-class**: the class that inherits properties and methods from another class, to fix ideas this is often called a child class;
- **super-class**: a class that is extended and that provides the basis for other classes to which it provides basic properties and methods, is often also called the parent class.
- Based on these concepts we define the concept of an animal with classes and then we create a dog:

# 4. INHERITANCE (FLUTTER)

```dart
// Base class
class Vehicle {
  String color;
  String model;

  Vehicle(this.color, this.model);

  String getInfo() {
    return 'Vehicle Model: $model, Color: $color';
  }
}

// Subclass
class Car extends Vehicle {
  int speed;

  Car(String color, String model) : speed = 0, super(color, model);

  String accelerate(int increment) {
    speed += increment;
    return 'The $color $model accelerates to $speed km/h.';
  }
}
```

```dart
class VehicleInfo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Create an instance of the Car class
    Car myCar = Car('Red', 'Toyota');

    // Get car information
    String carInfo = myCar.getInfo();
    String speedInfo = myCar.accelerate(20); // Accelerate the car by 20 km/h

    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Text(
            carInfo,
            style: TextStyle(fontSize: 20),
          ),
          SizedBox(height: 20),
          Text(
            speedInfo,
            style: TextStyle(fontSize: 20),
          ),
        ],
```



# Explanation

1. **Vehicle Class (Base Class):**
   - Contains two properties: `color` and `model`.
   - Has a constructor to initialize these properties.
   - Includes a method `getInfo()` that returns a string with the vehicle's information.
2. **Car Class (Subclass):**
   - Inherits from the `Vehicle` class using the `extends` keyword.
   - Adds a property `speed` and initializes it to `0`.
   - Includes a method `accelerate(int increment)` that increases the speed and returns a message.
3. **VehicleInfo Widget:**
   - Creates an instance of the `Car` class with specified color and model.
   - Calls the `getInfo()` method to get the vehicle's information.
   - Calls the `accelerate()` method to simulate the car accelerating by 20 km/h.
   - Displays the information using `Text` widgets within a `Column` for vertical arrangement.

## 4. INHERITANCE (CONT.)

# 5. POLYMORPHISM


Polymorphism is achieved through inheritance and it represents the ability of an object to copy the behavior of another object.

It means that one object can have multiple forms.

subclasses or child classes usually override instance methods, getters and setters. We can use @override to indicate that we are overriding a member.

Dart doesn't allow overloading. To overcome this we can use argument definitions like optional and positional.

# 5. POLYMORPHISM

```dart
// Base class
abstract class Animal {
  String makeSound();
}

// Subclass Dog
class Dog extends Animal {
  @override
  String makeSound() {
    return 'Woof!';
  }
}

// Subclass Cat
class Cat extends Animal {
  @override
  String makeSound() {
    return 'Meow!';
  }
}
```

```dart
class AnimalSounds extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Create a list of animals
    List<Animal> animals = [Dog(), Cat()];

    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: animals.map((animal) {
          return Text(
            'The animal makes: ${animal.makeSound()}',
            style: TextStyle(fontSize: 20),
          );
        }).toList(),
      ),
    );
  }
}
```

Polymorphism Example

The animal makes: Woof!
The animal makes: Meow!

# 5. POLYMORPHISM

Explanation

1. **Animal Class (Base Class):**
   - An abstract class `Animal` is created with a method `makeSound()`. This method will be implemented by subclasses.
2. **Dog Class (Subclass):**
   - The `Dog` class extends the `Animal` class and overrides the `makeSound()` method to return "Woof!".
3. **Cat Class (Subclass):**
   - The `Cat` class also extends the `Animal` class and overrides the `makeSound()` method to return "Meow!".
4. **AnimalSounds Widget:**
   - A list of `Animal` objects (containing a `Dog` and a `Cat`) is created.
   - The `map` function is used to call `makeSound()` on each animal in the list, demonstrating polymorphism.
   - The results are displayed using `Text` widgets within a `Column`.

Polymorphism Example

The animal makes: Woof!
The animal makes: Meow!

# ABSTRACTION IN ACTION

- **Abstraction** is the concept of hiding the complex implementation details of a system and exposing only the necessary and relevant features to the user.
- It allows developers to focus on interactions at a high level without needing to understand the intricate workings of the components.

**Abstract Classes:**

- An **abstract class** serves as a blueprint for other classes. It cannot be instantiated on its own and often contains one or more **abstract methods** that must be implemented by any subclass.
- **Abstract** classes can also have **concrete methods** (methods with implementation) and attributes.
- An **abstract class** can serve as an **interface** for other classes. This means that any class that implements the abstract class must provide implementations for all of its abstract methods.
- Unlike some other programming languages, **Dart** does not have a separate keyword for interfaces. Instead, **all classes** can act as **interfaces**, but **abstract classes** are a common way to define a contract.

```dart
// Abstract class
abstract class Device {
  // Abstract method to turn on the device
  void turnOn();
}

// Subclass Laptop
class Laptop implements Device {
  @override
  void turnOn() {
    print('Laptop is now ON.');
  }
}

// Subclass Smartphone
class Smartphone implements Device {
  @override
  void turnOn() {
    print('Smartphone is now ON.');
  }
}
```

# ABSTRACTION IN ACTION

## Explanation

1. **Device Class (Abstract Class):**
   - The `Device` class is an abstract class with an abstract method `turnOn()`. This method will be implemented by subclasses.
2. **Laptop Class (Subclass):**
   - The `Laptop` class implements the `Device` interface and provides its version of the `turnOn()` method, which prints a message indicating that the laptop is now on.
3. **Smartphone Class (Subclass):**
   - The `Smartphone` class also implements the `Device` interface and provides its version of the `turnOn()` method.
4. **DeviceInfo Widget:**
   - In the `DeviceInfo` widget, instances of `Laptop` and `Smartphone` are created.
   - The `turnOn()` method is called for each device to demonstrate polymorphism.
   - The output displays the types of devices on the screen.

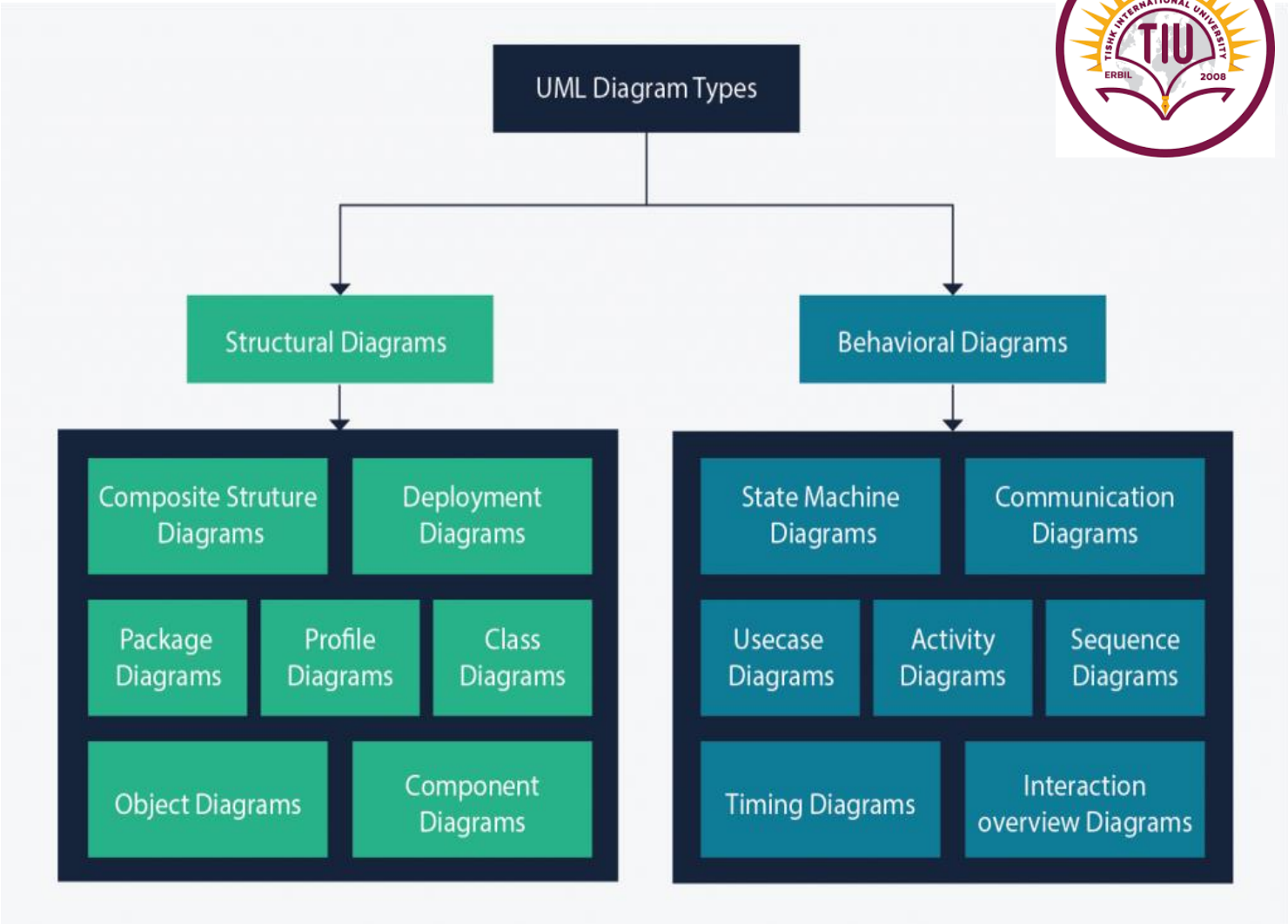
```dart
class DeviceInfo extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Create instances of Laptop and Smartphone
    Device myLaptop = Laptop();
    Device mySmartphone = Smartphone();

    // Call the turnOn method on each device
    myLaptop.turnOn();
    mySmartphone.turnOn();

    return Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Text(
            'Laptop: ${myLaptop.runtimeType}',
            style: TextStyle(fontSize: 20),
          ),
          Text(
            'Smartphone: ${mySmartphone.runtimeType}',
            style: TextStyle(fontSize: 20),
          ),
```

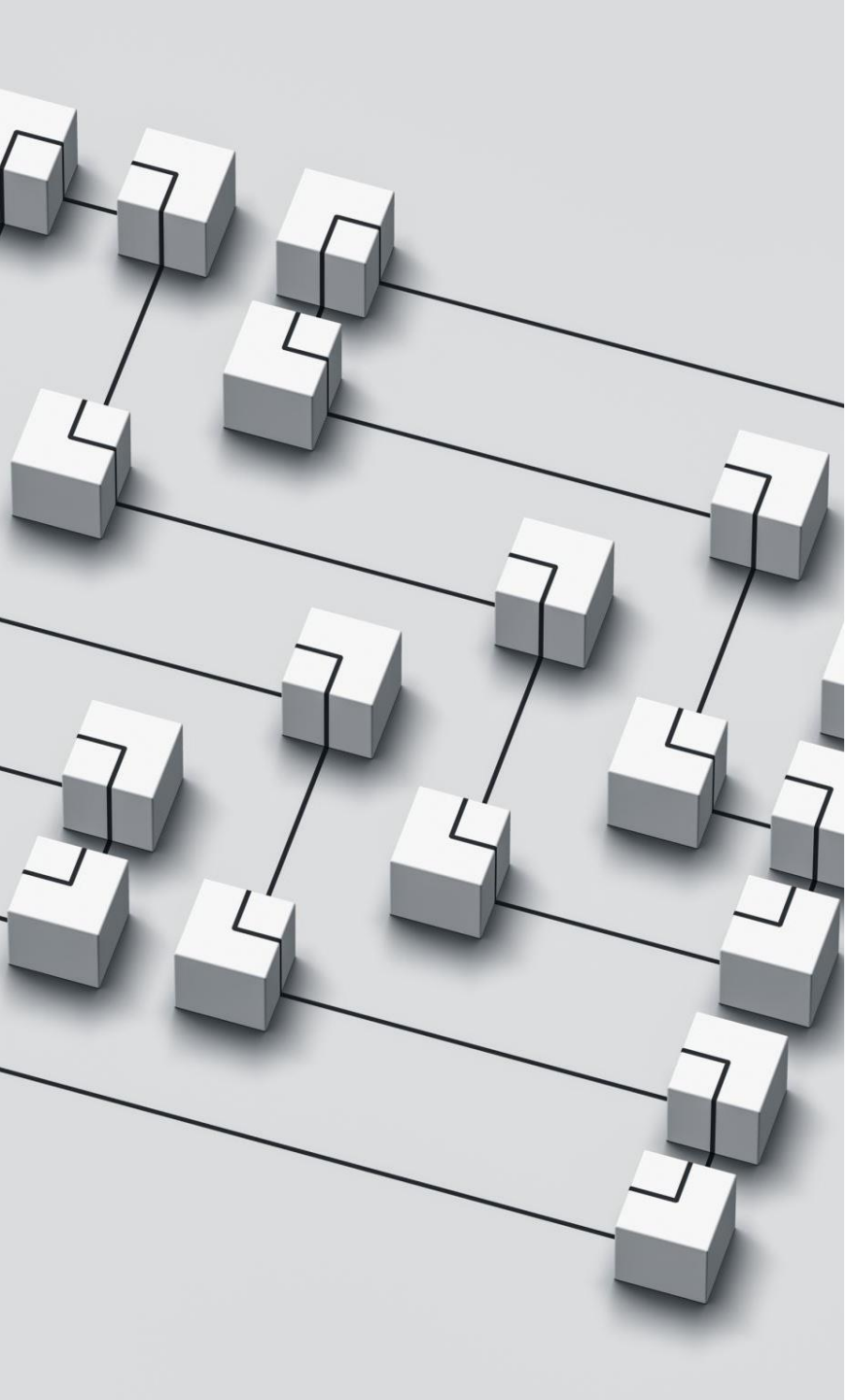
Abstraction Example

Laptop: Laptop
Smartphone: Smartphone

# UML DIAGRAM

- UML stands for **U**nified **M**odeling **L**anguage. It's a rich language to model software solutions, application structures, system behavior and business processes. There are **14 UML diagram types** to help you model these behaviors.
- List of UML Diagram Types
- So what are the different UML diagram types? There are two main categories; **structure diagrams** and **behavioral diagrams**.





# CLASS DIAGRAM

- Class diagram is the backbone of object-oriented modeling - it shows how different entities (people, things, and data) relate to each other.
- In other words, it shows the **static** structures of the system.
- A class diagram describes the attributes and operations of a class and also the constraints imposed on the system.
- Class diagrams are widely used in the modeling of object-oriented systems because they are the only UML diagrams that can be mapped directly to object-oriented languages.

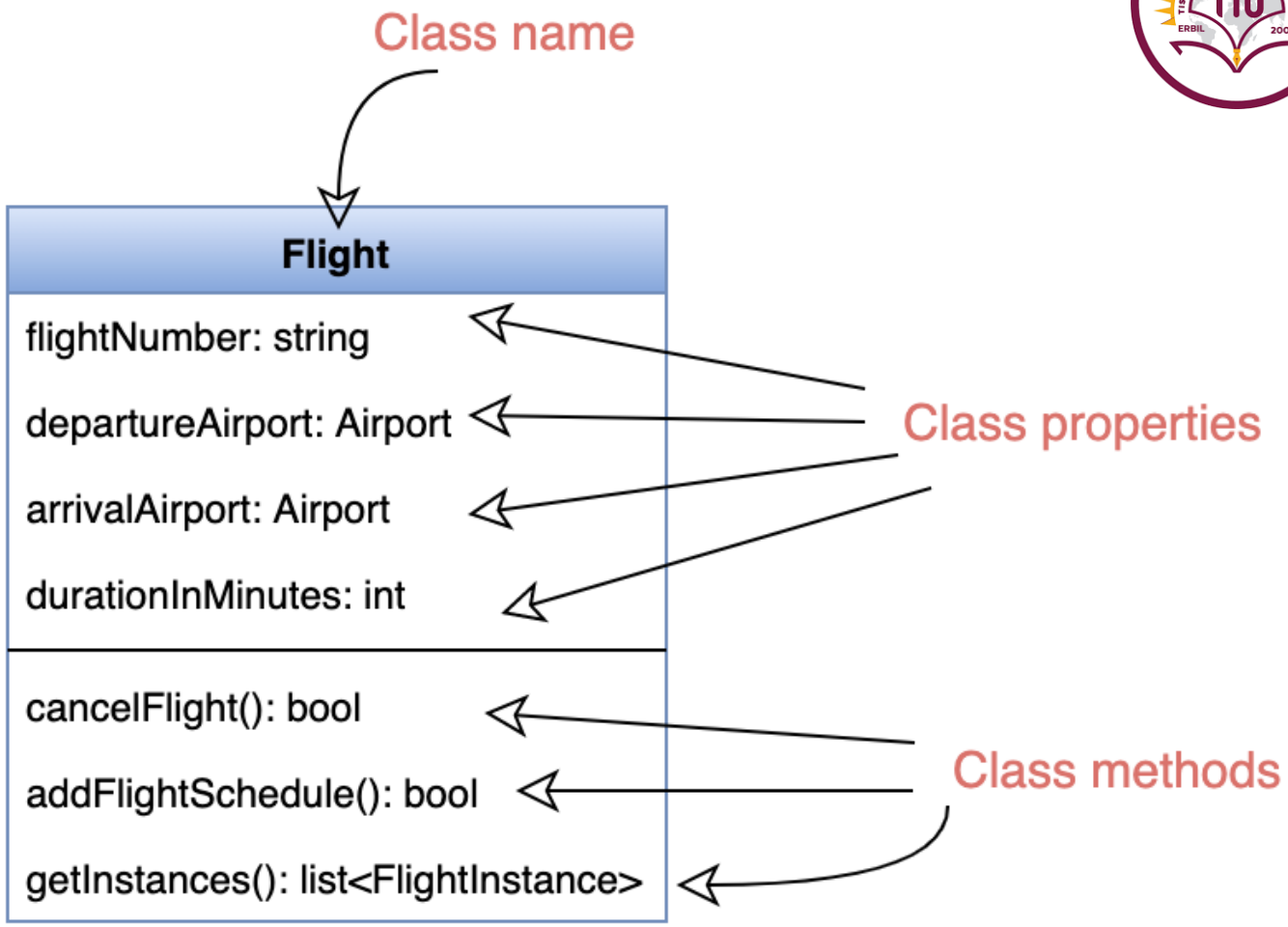
The purpose of the class diagram can be summarized as:

- Analysis and design of the static view of an application;
- To describe the responsibilities of a system;
- To provide a base for component and deployment diagrams; and,
- Forward and reverse engineering.

# CLASS DIAGRAM (CONT.)

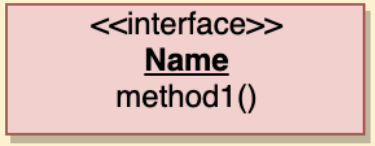
A class is depicted in the class diagram as a rectangle with three horizontal sections, as shown in the figure below.

- The upper section shows the class's name (Flight),
- the middle section contains the properties of the class,
- and the lower section contains the class's operations (or "methods").

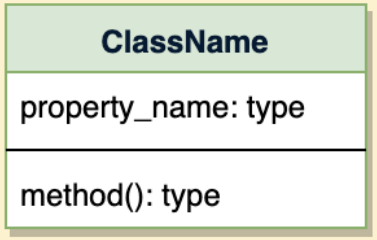


# CLASS DIAGRAM (CONT.)

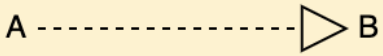
## UML conventions



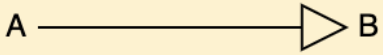

**Interface**: Classes implement interfaces, denoted by Generalization.



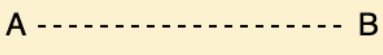

**Class**: Every class can have properties and methods.
Abstract classes are identified by their *Italic* names.



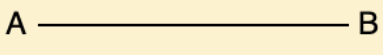

**Generalization**: A implements B.



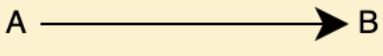

**Inheritance**: A inherits from B. A "is-a" B.



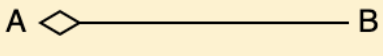

**Use Interface:** A uses interface B.



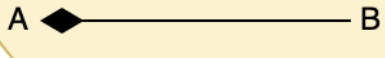

**Association**: A and B call each other.



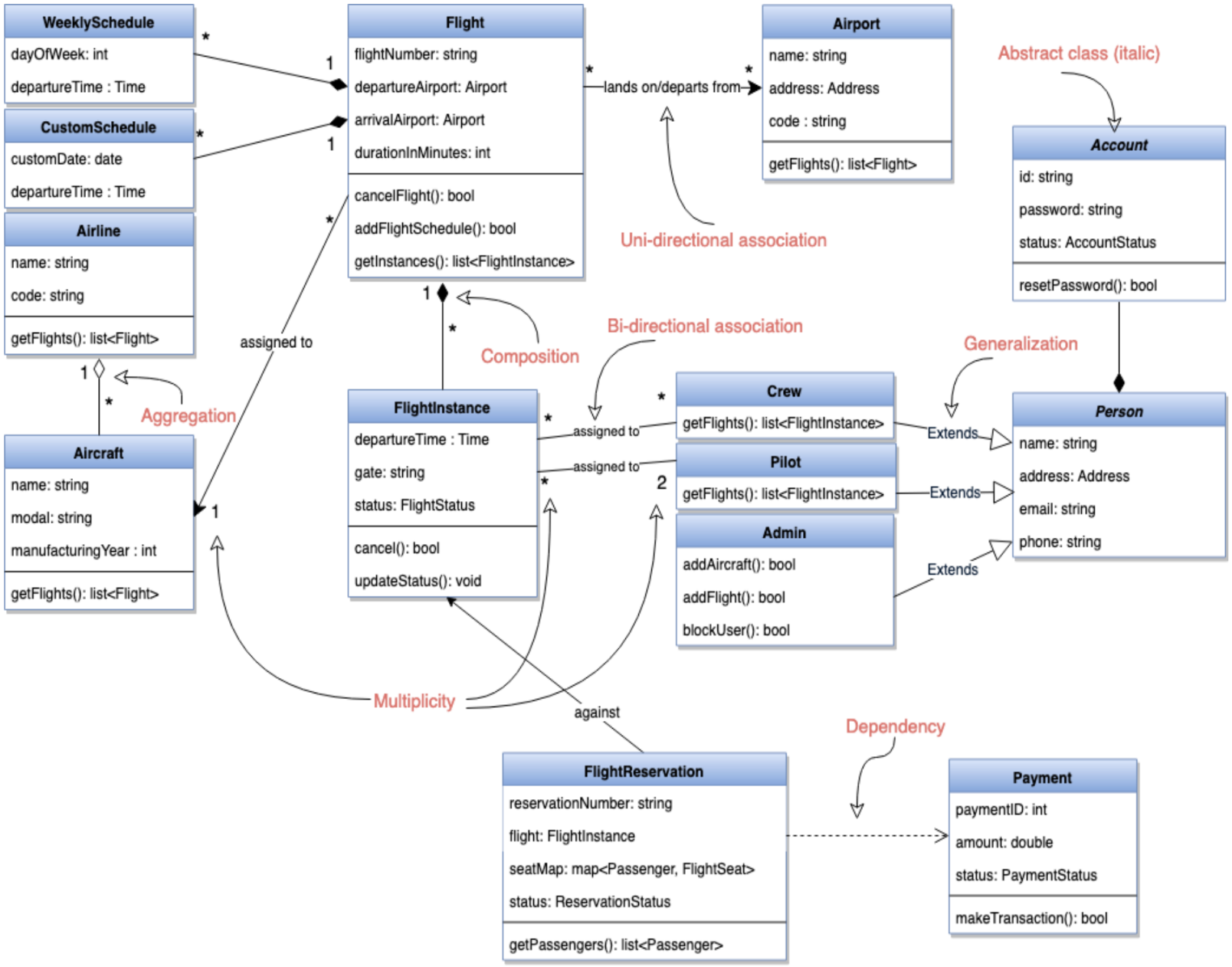

**Uni-directional Association**: A can call B, but not vice versa.




**Aggregation**: A "has-an" instance of B. B can exist without A.

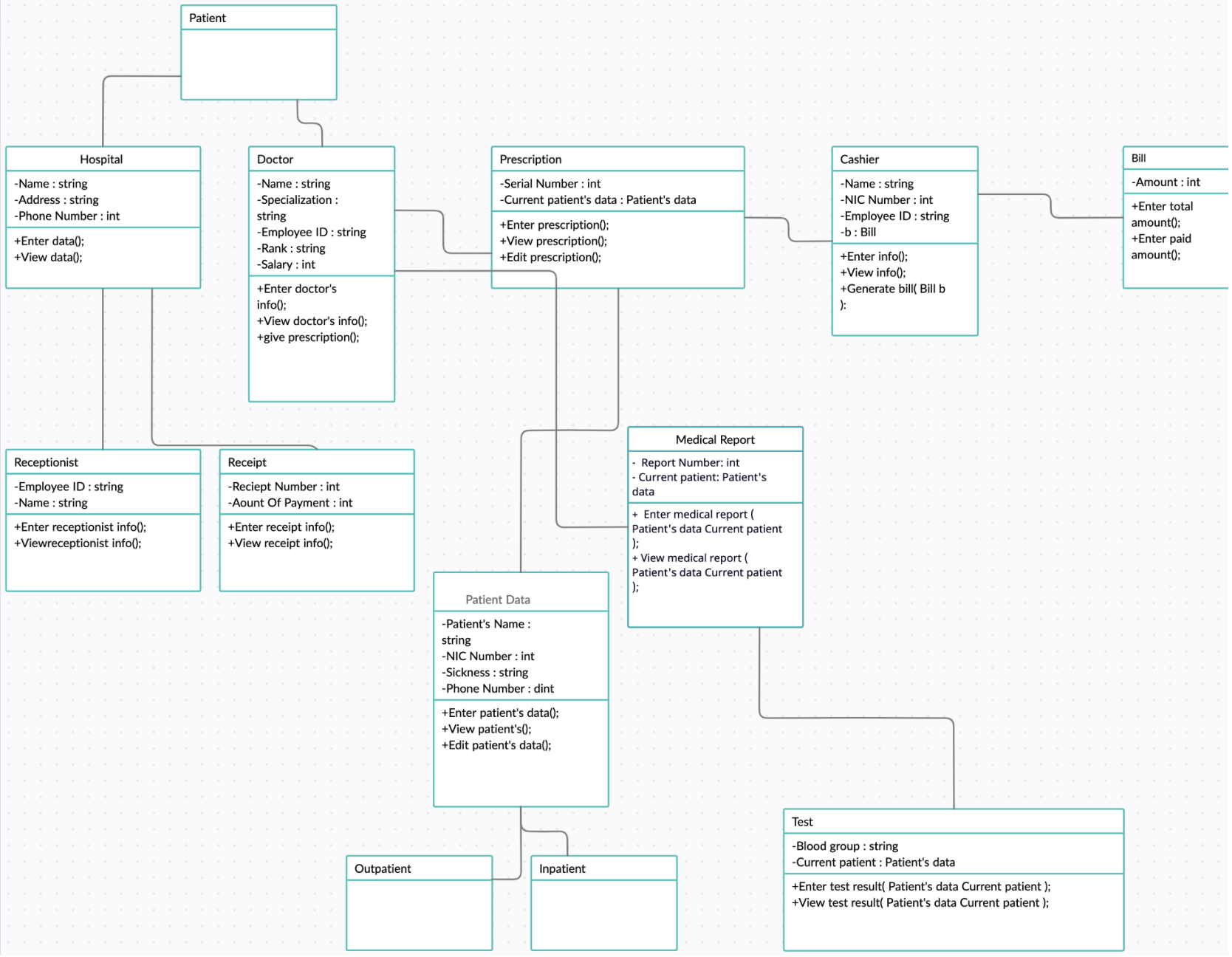



**Composition**: A "has-an" instance of B. B cannot exist without A.




# CLASS DIAGRAM (CONT.)

## SAMPLE CLASS DIAGRAM FOR FLIGHT RESERVATION SYSTEM




# CLASS DIAGRAM EXAMPLE: HOSPITAL MANAGEMENT SYSTEM

# SUMMARY


In this lecture notes we learned about many important aspects:

- Key facts about OOP
- Trending Programming Languages 2024 and our Dart ☺
- OOP languages and how popular they are!
- OOP Features in Flutter
- UML Diagram

Finally, OOP features in Dart with some examples

*Please note that you if you needed to remember anything about Dart, you can go back to your Programming 2 or OOP lecture notes or use the ChatGPT for more details.*