OBJECT-ORIENTED PROGRAMMING I
IT 311

IT DEPT.

TIU

3RD GRADE

Lect. Mohammad Salim

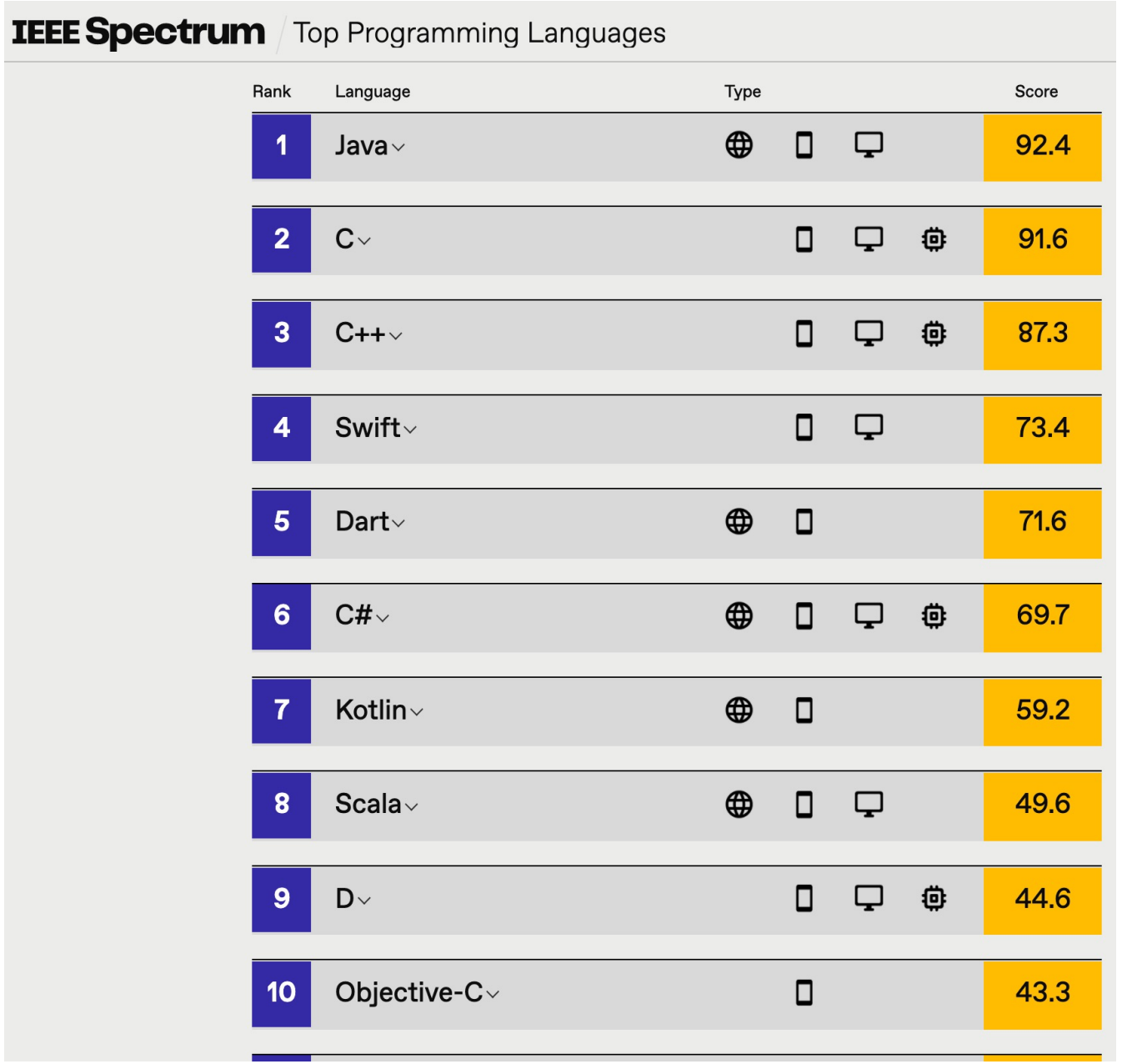
CONTENTS

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.



IEEE Spectrum / Top Programming Languages

Rank	Language	Type	Score
1	Java		92.4
2	C		91.6
3	C++		87.3
4	Swift		73.4
5	Dart		71.6
6	C#		69.7
7	Kotlin		59.2
8	Scala		49.6
9	D		44.6
10	Objective-C		43.3

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.

Object-oriented programming

CLASS
Human

OBJECTS
Name

PROPERTIES
Email
Address

METHODS
Verify
Send mail

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

```
class class_name
{
   <constructors>
   <functions>
   <fields>
}
```

Example for a Class

```
class example
{
 string myname = "OpenGenus Foundation" ;
 void disp()
  {
   print(name) ;
  }
}
```

Here

example is the class name

string is the field of class

void disp() is the function of the class

2. OBJECTS

Object is an entity. Objects are declared to access functions and data declared in a class.

Synatx for declaring an object :

```
var object_name = new class_name(arguments);
```

Example for Object declaration

```
var name = new example(arguments);
```

Here ,
object name is name
new creates a new object
example is the class name

For accessing the class properties and methods, we use . operator.

3. DATA ENCAPSULATION

- Data Encapsulation is binding data and functions that use data into one unit. It is also referred to as data hiding and information hiding.
  Unlike C++ and Java , Dart does not have keywords for restricting access like private , public and protected.
- 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**.

Syntax

```
_identifier
```

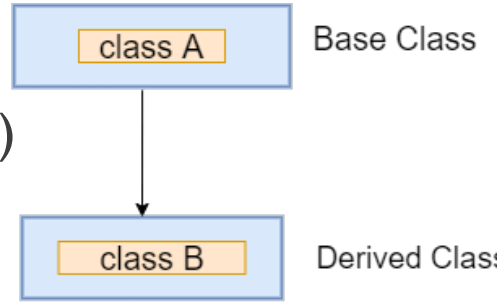
Example

```
library loggerlibrary;
void _log(message)
{
   print("Log method called in the loggerlibrary message:$message");
}
```

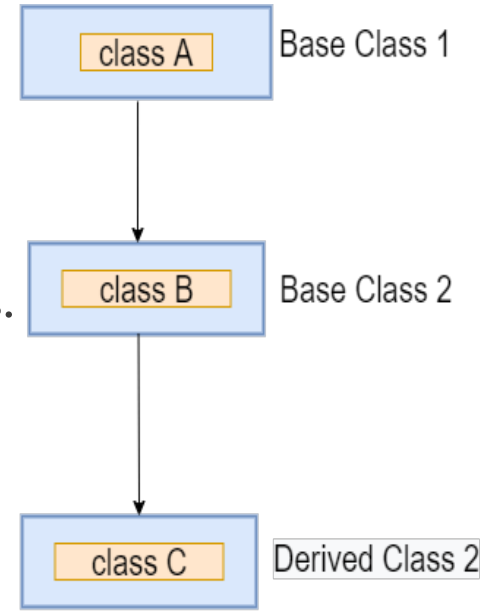
In the above example we have defined a library with a private function.

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.



Single Inhetitance



Multilevel 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**.

4. INHERITANCE (CONT.)

Example for Inheritance (Single Level)

```
void main()
{
   var obj = new model();
   obj.price();
}
class car
{
   void price()
   {
      print(" price of car model in the car class");
   }
}
class model extends car{}
```

Output

```
Price of car model in the car class
```


Do note that a child class inherits all the methods from the parent class except the constructors.

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

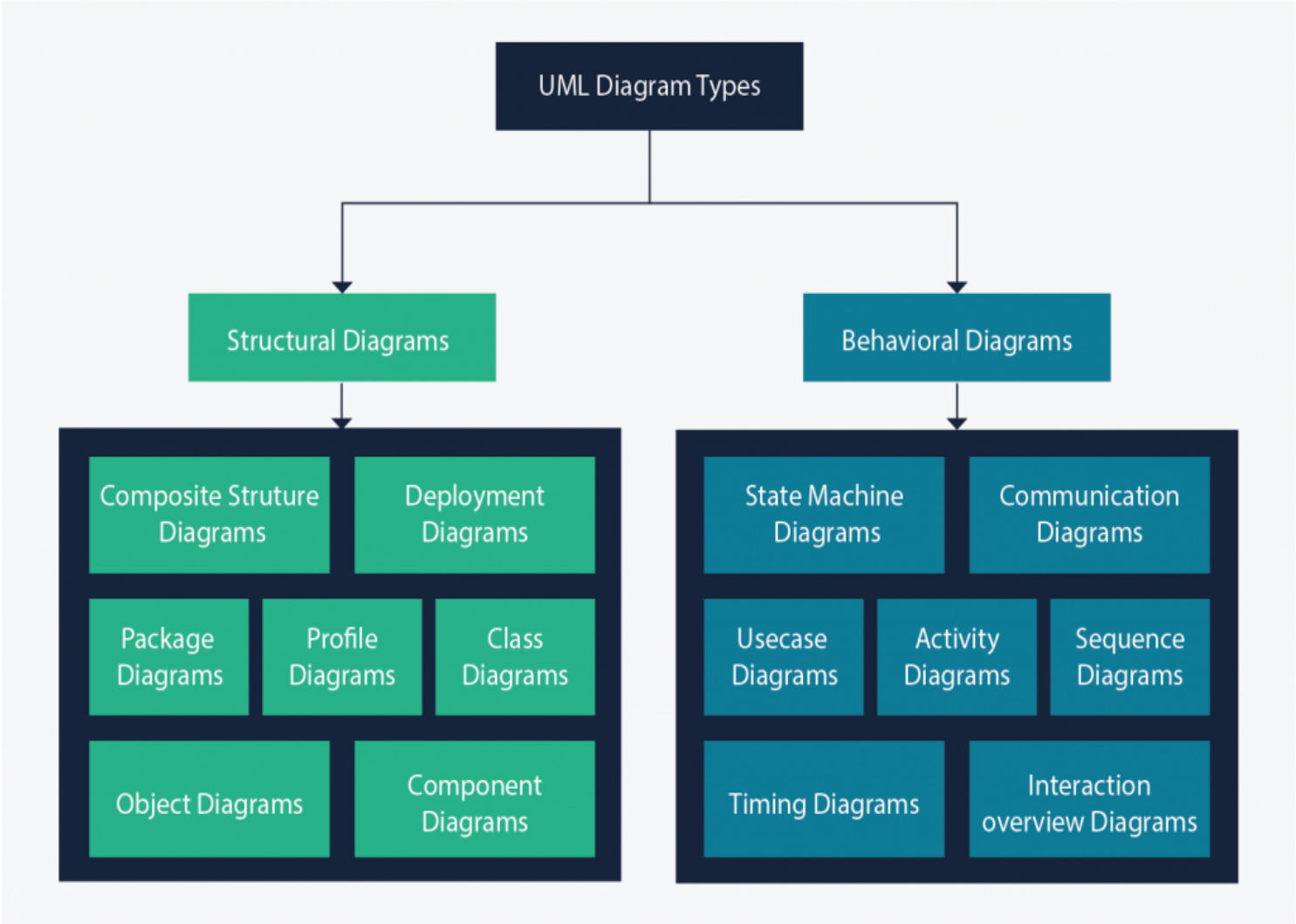
```
class Car extends Vehicle
{
  Car()
  {
    this.topspeed = 240;
    this.name = "Car";
  }

  void EngineStart()
  {
    print('Engine Started');
  }
}

main()
{
  car model ;
  model = new Lights(73);
  model.turnOn();
  model.goForward();
  model.turnOff();

  model = new Car();
  model.turnOn();
  model.goForward();
  model.turnOff();
}
```

here the object - "model" has multiple forms

With this article at OpenGenus, you must have a general idea of OOP concepts in Dart. Enjoy.

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.

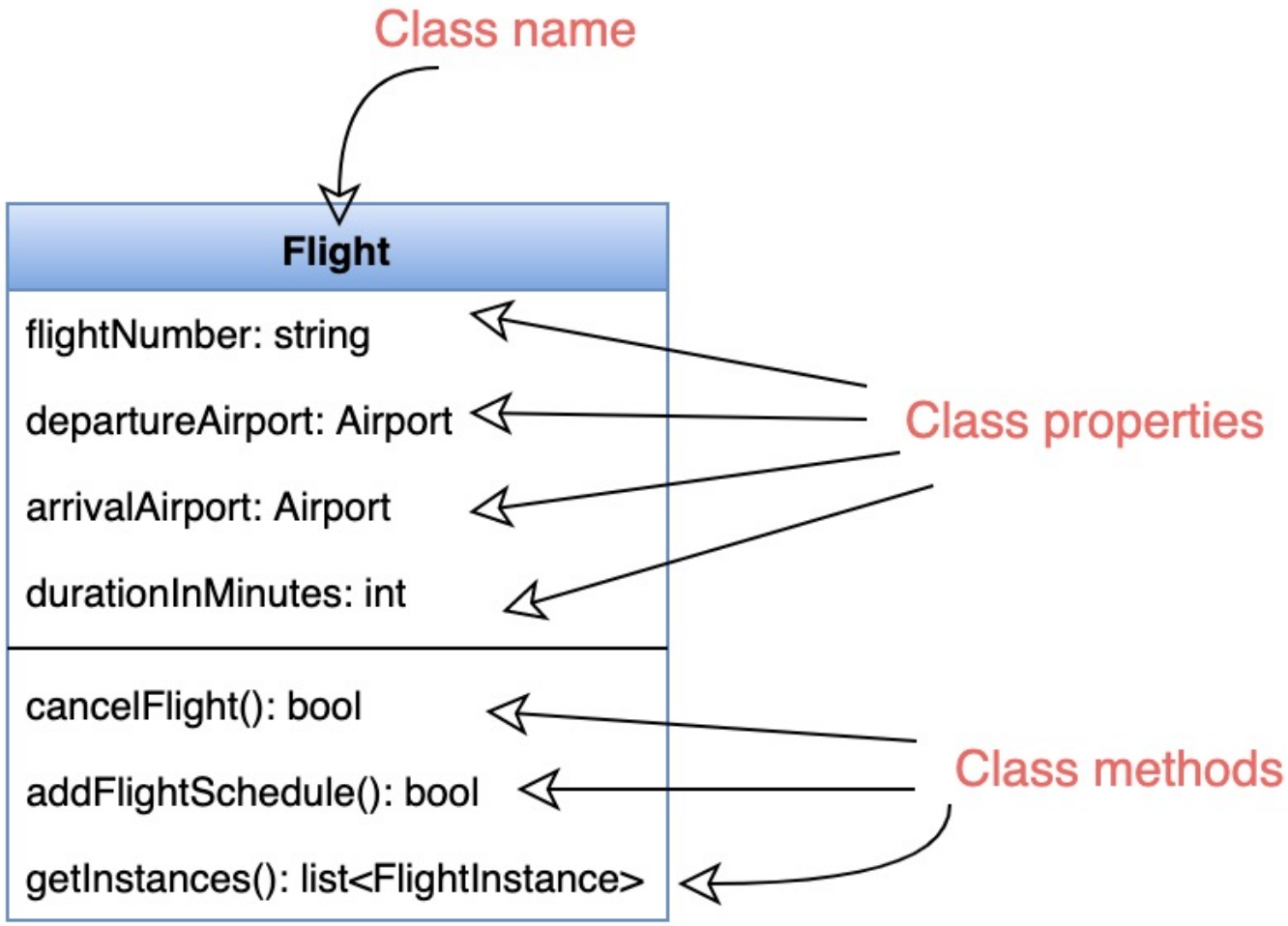
The purpose of the class diagram can be summarized as:

- 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.)

- These are the different types of relationships between classes:

Association: If two classes in a model need to communicate with each other, there must be a link between them. This link can be represented by an association. Associations can be represented in a class diagram by a line between these classes with an arrow indicating the navigation direction.

- By default, associations are always assumed to be bi-directional; this means that both classes are aware of each other and their relationship. In the diagram below, the association between Pilot and FlightInstance is bi-directional, as both classes know each other.
- By contrast, in a uni-directional association, two classes are related - but only one class knows that the relationship exists. In the below example, only Flight class knows about Aircraft; hence it is a uni-directional association

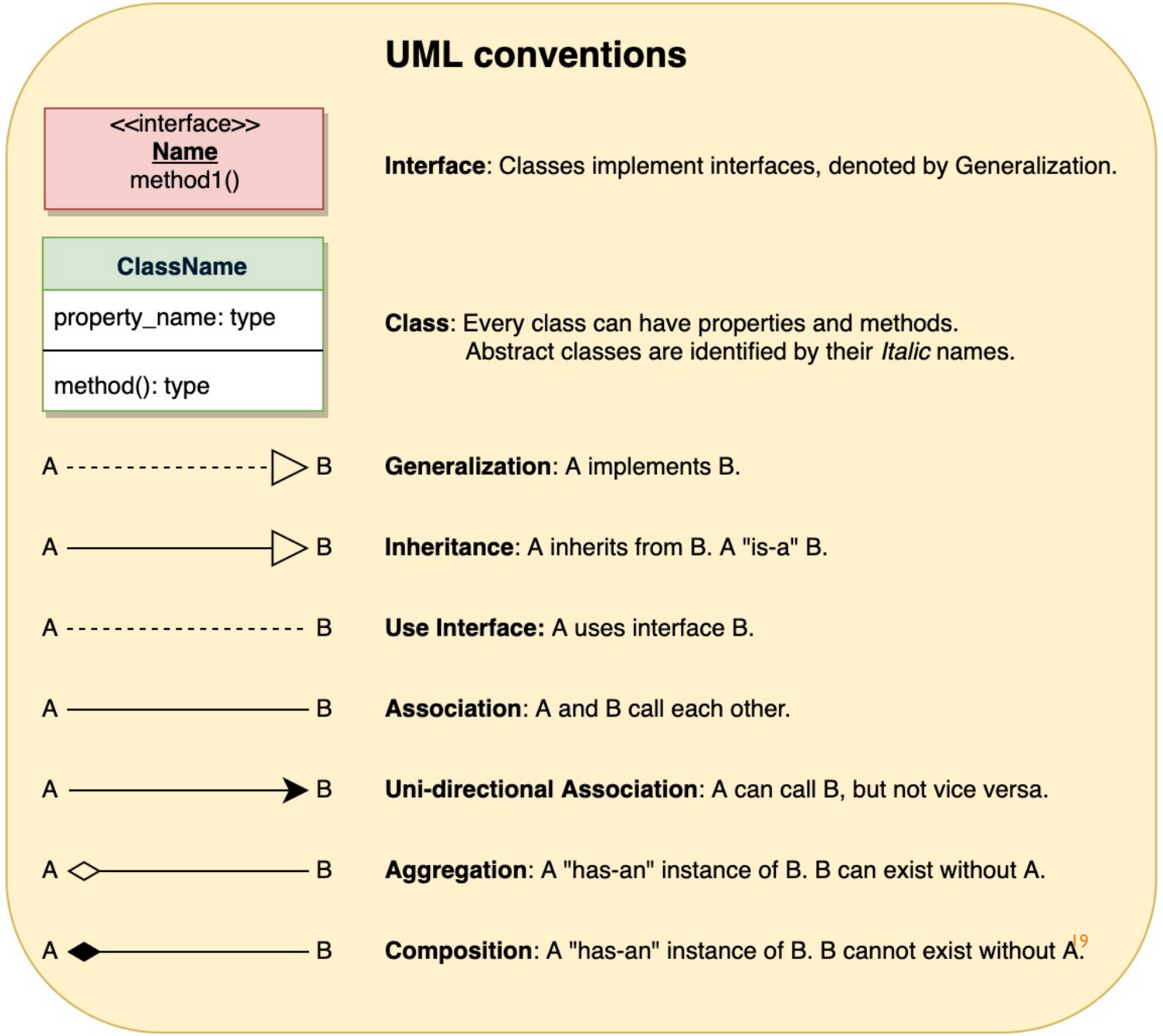
Multiplicity Multiplicity indicates how many instances of a class participate in the relationship. It is a constraint that specifies the range of permitted cardinalities between two classes. For example, in the diagram below, one FlightInstance will have two Pilots, while a Pilot can have many FlightInstances. A ranged multiplicity can be expressed as "0…*" which means "zero to many" or as "2…4" which means "two to four".

- We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association. The below diagram, demonstrates that a FlightInstance has exactly two Pilots but a Pilot can have many FlightInstances.

CLASS DIAGRAM (CONT.)

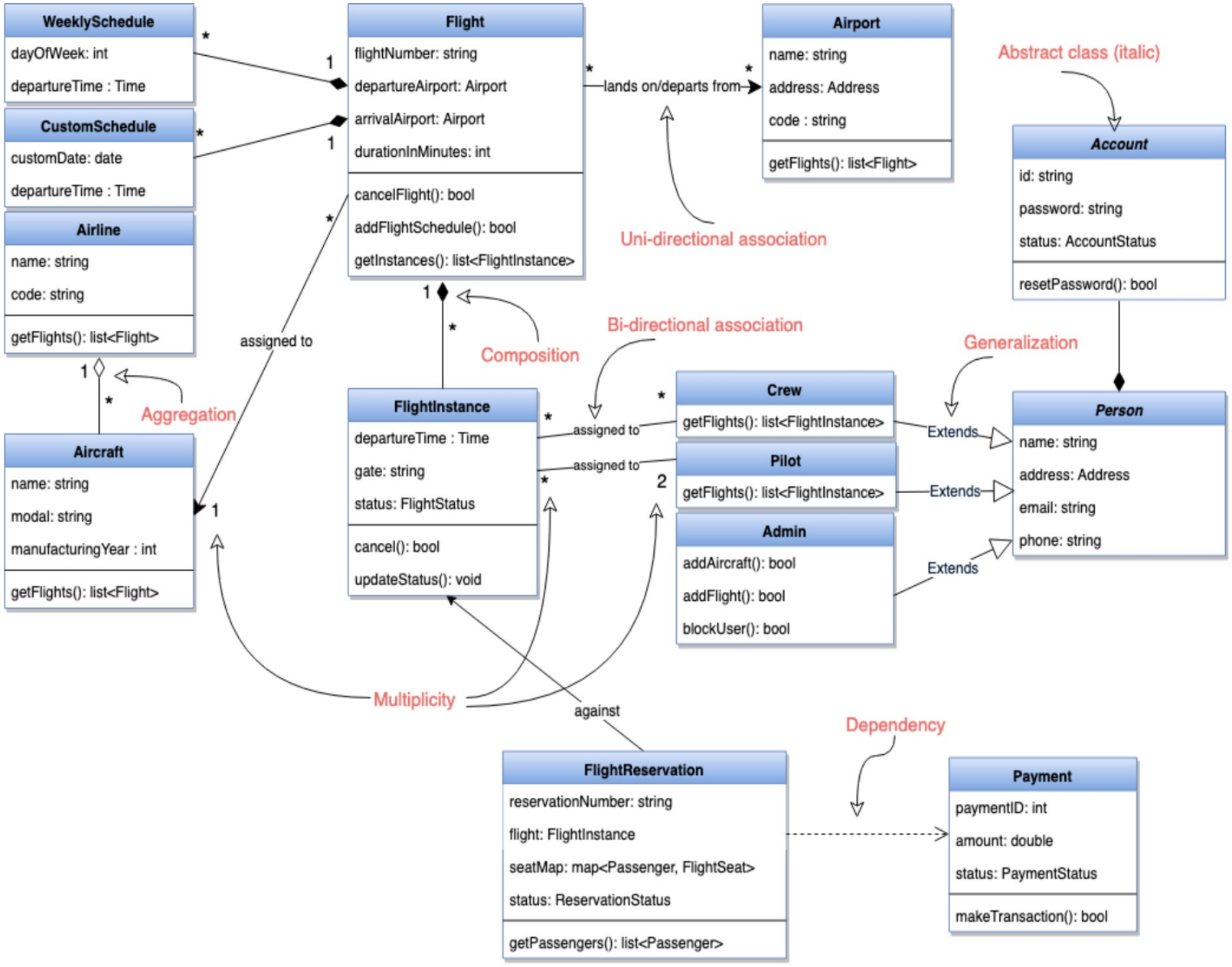
- **Aggregation:** Aggregation is a special type of association used to model a "whole to its parts" relationship. In a basic aggregation relationship, the lifecycle of a PART class is independent of the WHOLE class's lifecycle. In other words, aggregation implies a relationship where the child can exist independently of the parent. In the above diagram, Aircraft can exist without Airline.
- **Composition:** The composition aggregation relationship is just another form of the aggregation relationship, but the child class's instance lifecycle is dependent on the parent class's instance lifecycle. In other words, Composition implies a relationship where the child cannot exist independent of the parent. In the above example, WeeklySchedule is composed in Flight which means when Flight lifecycle ends, WeeklySchedule automatically gets destroyed.
- **Generalization:** Generalization is the mechanism for combining similar classes of objects into a single, more general class. Generalization identifies commonalities among a set of entities. In the above diagram, Crew, Pilot, and Admin, all are Person.
- **Dependency:** A dependency relationship is a relationship in which one class, the client, uses or depends on another class, the supplier. In the above diagram, FlightReservation depends on Payment.
- **Abstract class:** An abstract class is identified by specifying its name in *italics*. In the above diagram, both Person and Account classes are abstract classes.

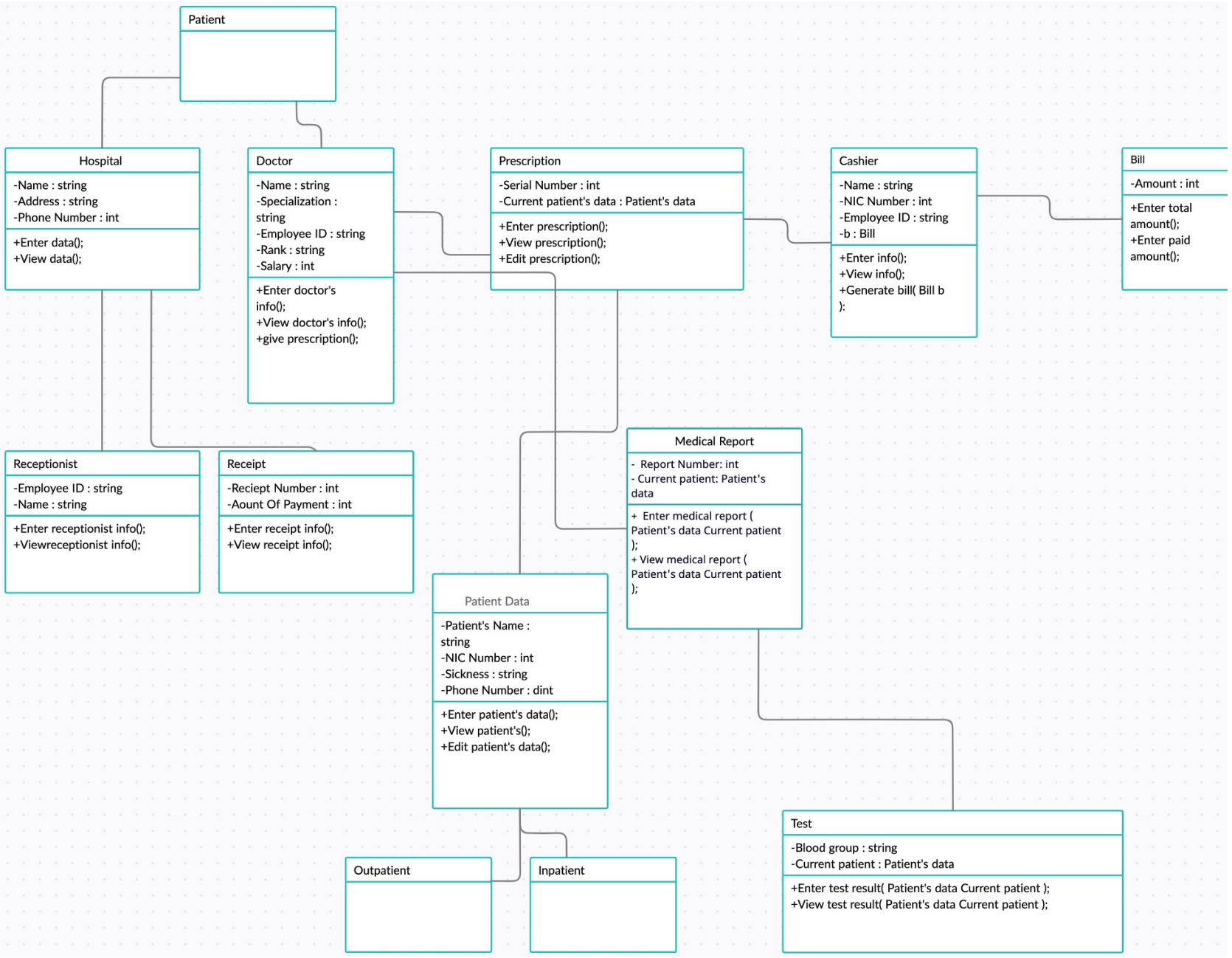
CLASS DIAGRAM (CONT.)



CLASS DIAGRAM (CONT.)

SAMPLE CLASS DIAGRAM FOR FLIGHT RESERVATION SYSTEM





CLASS DIAGRAM EXAMPLE: HOSPITAL MANAGEMENT SYSTEM

CLASSES AND OBJECTS

- 1- CLASSES AND OBJECTS
- 2- ABSTRACTION IN ACTION
- 3- ENCAPSULATION IN ACTION
- 4- INHERITANCE IN ACTION
- 5- POLYMORPHISM IN ACTION

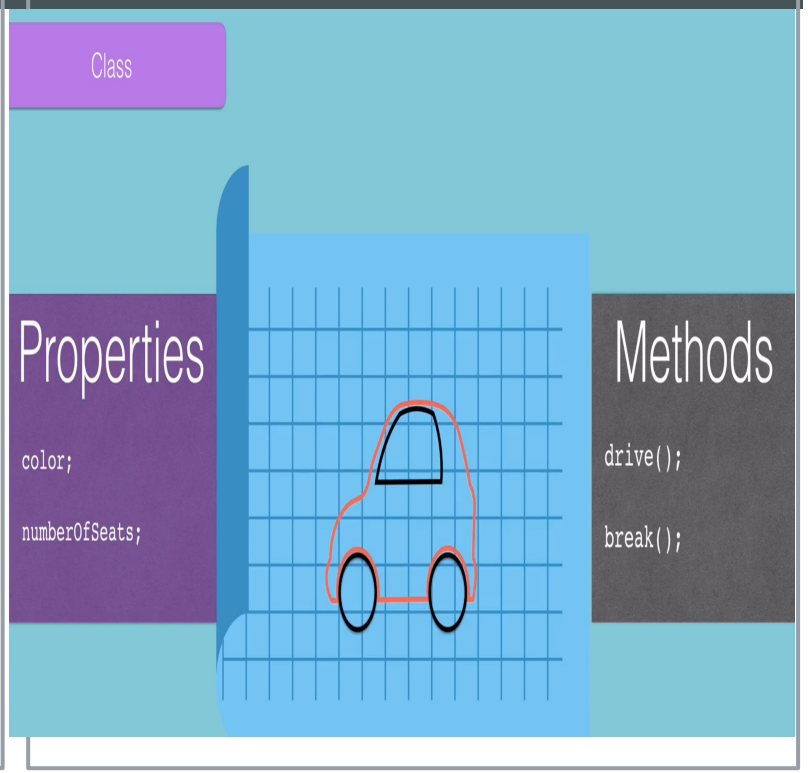
```
class Car {

  int numberOfDoors = 5;

  void drive() {
    print('wheels start turning');
  }

}
```

Creating the Class

```
class Car {
  int numberOfSeats = 5;
  void drive() {
    print('wheels start turning');
  }
}
```

Creating an Object from the Class

```
Car myCar = Car();
```



CLASSES AND OBJECTS

DART

```dart
void main() {

  Human jenny = Human(startingHeight: 15);

  print(jenny.height);

  jenny.talk('Why is the sky blue');

}


class Human {

  double height;
  int age = 0;

  Human({double startingHeight}) {
    height = startingHeight;
  }

  void talk(String whatToSay) {
    print(whatToSay);
  }

}
```

CLASSES AND OBJECTS

ABSTRACTION IN ACTION

- Abstraction is the first principle of OOP and like many other programming languages, it's a way to reduce code complexity by dividing functionality into different "chunks" and exposing only essential functionality to the outside world.
- Abstraction allows you to focus on what a class does instead of how it does it
- In other words, when you instantiate a **class** you only have to worry about the **methods** it provides, **parameters** these methods receive, and the **outputs** it returns.
- All the actual implementation parts must not be known externally so if you want to change them, all the code dependent on that class would not be affected by the change.
- Let's make an example with a simple Square class:

ABSTRACTION IN ACTION

COPY

```
class Square {
  int side;

  Square({this.side});

  int getPerimeter(){
    return side+side+side+side;
  }
}
```

I defined a **Square** class and I said that it has a property named **side** and a method that returns its perimeter. So if I create a Square object with a side of 10, by calling **getPerimeter()** I can get its perimeter:

COPY

```
Square mySquare = Square(side:10);
print(mySquare.getPerimeter());
```

Obviously, the code prints 40. Let's suppose that now I found a super optimized way to calculate the perimeter with this new function:

COPY

```
int getPerimeter(){
    return side*4;
}
```

Now if I want to instantiate another Square object and prints its perimeter I don't have to change anything because I have "abstracted" the perimeter calculation functionality, I hid the details on how I calculated perimeter focusing only on giving the perimeter calculation functionality.

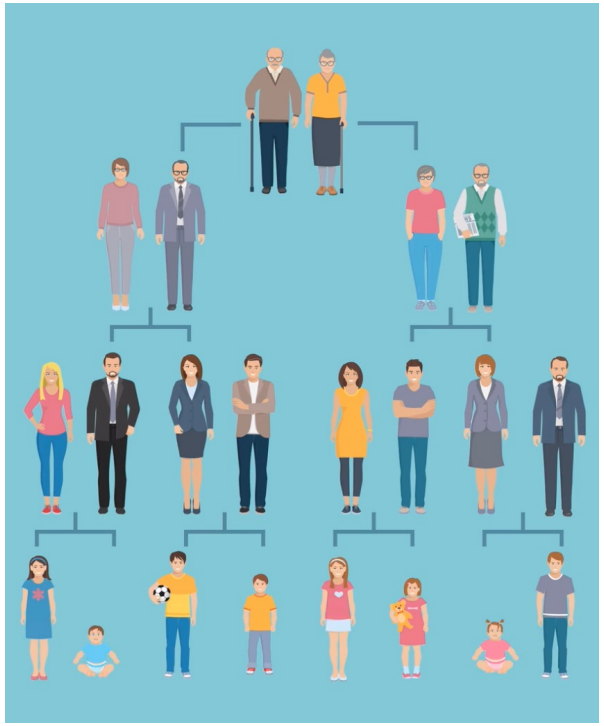
ENCAPSULATION IN ACTION

- This is the second principle of OOP and it refers to the ability of an object to **hide** its data or its state and allow access to its properties only through **particular methods**.
- In other languages, this is done by defining the property **private** and the methods **public**.
- So if someone instantiates a class of a certain type he cannot access directly to properties but he must use methods to change property values. A simple example to show this concept is:
- If managed correctly, it allows you to see the object as a black-box, with which the interaction takes place only and only through the methods defined by the class.
- Encapsulation and Abstraction are very connected to each other because they allow you to expose functionality outside the class hiding the implementation details.

```
1  class Human {
2    //With the underscore, I can define a variable private
3    // so from the outside this variable cannot be changed directly
4    int? _superSecretVariableForTheAge;
5
6    Human();
7
8    //I create two methods called getter and setter to modify
9    // properties inside my class
10   void setAge(int newAge){
11      this._superSecretVariableForTheAge = newAge;
12   }
13   int getAge(){
14     return _superSecretVariableForTheAge!;
15   }
16 }
17 void main() {
18
19   Human mySelf = Human();
20   mySelf.setAge(33);
21   print(mySelf.getAge());
22
23   //I cannot acces directly the property with this line of code
24   mySelf._superSecretVariableForTheAge = 44;
25
26 }
```

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:



INHERITANCE IN ACTION

```
//   myNormalCar.drive();

  ElectricCar myTesla = ElectricCar();

  myTesla.drive();

  myTesla.recharge();

}


class Car {
  int numberOfSeat = 5;

  void drive() {
    print('wheels turn.');
  }
}

class ElectricCar extends Car {
  int batteryLevel = 100;

  void recharge() {
    batteryLevel = 100;
  }
}
```

DART

```
void main() {

  Car myNormalCar = Car();

  print(myNormalCar.numberOfSeat);

  myNormalCar.drive();

  ElectricCar myTesla = ElectricCar();

  myTesla.drive();

}


class Car {
  int numberOfSeat = 5;

  void drive() {
    print('wheels turn.');
  }
}
```

- In the example, you can see that I create **ElectricCar** with a class type **Car** that inherits from the **Car** class. By doing that I can make it **recharge** and give it a **batteryLevel** .Also we make it drive using the method defined in its parent class because all cars have the ability to drive and have numberOfSeats.
- Inheritance can be of the following three types: Single, Multiple and Multi-level. Dart supports Single Inheritance and Multi-level, so a class must extend from only one parent class and you can create multi-level relationships like this: *grandparent class -> parent class -> child class*.

```dart
class SelfDrivingCar extends Car {

  String destination;

  SelfDrivingCar(String userSetDestination) {
    destination = userSetDestination;
  }

  @override
  void drive() {
    super.drive();

    print('sterring towards $destination');
  }

}
```

```dart
void main() {
  SelfDrivingCar myWaymo = SelfDrivingCar('1 Hacker Way');

  myWaymo.drive();

}
```

Dart based on the type of the variables **ElectricCar** and **LelivatingCar** correctly run functions defined relatively in the **ElectricCar** and **LelivatingCar** classes.

DART

```dart
class Car {
  int numberOfSeat = 5;

  void drive() {
    print('wheels turn.');
  }
}

class ElectricCar extends Car {
  int batteryLevel = 100;

  void recharge() {
    batteryLevel = 100;
  }
}

class LevitatingCar extends Car {

  @override
  void drive() {
    print('glide forwards');
  }

}
```

POLYMORPHISM IN ACTION

- Polymorphism refers to the language's ability to use the correct method of a class based on the type of the variable at runtime and not at compile time.
- This way if you have a method defined in a super class and two child classes that override it, the language at runtime will execute the correct function code based on the type of the variable. Let's take an example to better understand:

SUMMARY

In this lecture notes we learned about many important aspects:

- Key facts about OOP
- Trending Programming Languages 2021 and situation of our Dart !
- OOP languages and how popular they are !
- OOP Features in general
- 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 lecture notes or use the Internet search engines for more details.