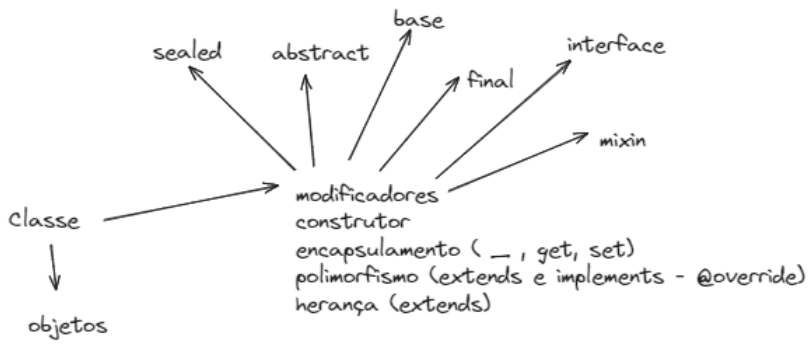
Pesquisa OOP com Dart

Programação orientada a objetos com Dart



CLASSE

Uma classe é uma estrutura que permite moldar objetos. Ela representa uma abstração de algo do mundo real.

OBJETO

Um objeto é uma instância de uma classe. A partir do molde da classe disponível, é possível criar várias representações daquilo que se deseja.

Em dart, é possível a criação de uma classe e objeto da seguinte maneira (muito similar a estrutura de outras linguagens de programação como Java e JavaScript):

```
/* criação de classe */

class Employee {
  int id;
  String name;

  void displayEmployeeDetails() {
    print("Employee ID: $id");
    print("Employee Name: $name");
  }
}
```

```
/* criação de objeto a partir da classe de Employee */

void main() {
    Employee emp = new Employee();
    emp.id = 1;
    emp.name = "John Doe";
    emp.displayEmployeeDetails();
}
```

Nesse sentido então, a classe abstrai o conceito do colaborador de uma empresa, indicando que todo colaborador irá compartilhar de mesmas características, como possuir um identificador e um nome. Portanto, essas características na orientação a objetos é o que chamamos de propriedades. Abaixo das propriedades temos os métodos que poderão ser utilizados para performar ações no objeto.

Ao criar um objeto a partir da classe, ou seja, no momento de criar um novo colaborador, é necessário instanciar a classe, alocando na variável de objeto. A partir desse momento, dependendo de como foi definido na classe, o novo colaborador tem acesso as propriedades e métodos. Como é possível verificar na criação, é definido um identificador e nome por meio das propriedades e é possível chamar o método, tudo por meio da variável que representa o objeto.

HERANÇA

Uma herança entre classes auxilia na reutilização de códigos e abstração. No exemplo da empresa, a classe Employee serve como base para colaboradores de um modo generalista. Porém, e se for necessário outras características mais específicas para outro tipo de colaborador? Nesse caso é possível criar uma nova classe, como por exemplo Manager, que irá estender a classe de Employee, ou seja, ele irá herdar as propriedades e métodos da classe Employee.

```
class Manager extends Employee {
  String department;

  void displayManagerDetails() {
    print("Manager Department: $department");
  }
}
```

Portanto, não somente o id e o nome, o funcionário do tipo Manager também possui um departamento.

ENCAPSULAMENTO

Funciona diferente de outras linguagens pois não possui as palavras chaves public, private e protected, porém ao utilizar _ antes de uma classe, propriedade, método, entre outros, é possível criar uma privacidade.

Em Dart para que se possa realizar um tipo de encapsulamento, são utilizados os métodos get e set de forma implícita. E, é possível criar novos métodos getters e setters de forma explícita para manipulação dos dados:

```
class Rectangle {
  double left, top, width, height;

  Rectangle(this.left, this.top, this.width, this.height);

  // Define two calculated properties: right and bottom.
  double get right => left + width;
  set right(double value) => left = value - width;
  double get bottom => top + height;
  set bottom(double value) => top = value - height;
}

void main() {
  var rect = Rectangle(3, 4, 20, 15);
  assert(rect.left == 3);
  rect.right = 12;
```

```
  assert(rect.left == -8);
}
```

Ao utilizar o método get, conseguimos realizar uma operação para retornar uma propriedade computada a partir das propriedades da classe, sem acessar as propriedades diretamente. O mesmo ocorre com o método set, em que conseguimos atualizar o valor de uma propriedade sem acessá-la diretamente.

POLIMORFISMO

Permite que os objetos possam tomar diferentes formas, pois permite que métodos de classes-mãe sejam alterados. A alteração ocorre com o identificador @override sobre o método.

```
class Manager extends Employee {
  String department;

  @override
  void displayEmployeeDetails() {
    super.displayEmployeeDetails();
    print("Manager Department: $department");
  }
}
```

No caso do exemplo do Manager, os objetos que forem instâncias de Manager poderão utilizar o método da classe de Employee dentro do método da classe filha. Pode-se notar então que há também uma reutilização de código, ao invés de criar um novo método separado para mostrar a informação do departamento, o que ficaria desconexo visto que já há um método para mostrar as informações, o polimorfismo permite que esses métodos possam ser “juntados/acoplados” e ao utilizar um método somente (displayEmployeeDetails) todas as informações serão mostradas.

OUTROS CONCEITOS

CONSTRUTOR

O construtor de uma classe serve como um “fabricador” de objetos, de uma forma mais rápida e concisa. Utilizando o mesmo exemplo da classe de Employee, supondo que houvessem muitas propriedades dentro da classe, porém para que fosse possível a criação de um objeto do tipo Employee fosse necessário somente algumas propriedades, nesse caso isso pode ser definido diretamente no construtor.

Para criar ele basta utilizar o mesmo nome da classe.

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

  Employee(this.id, this.name);

  void displayEmployeeDetails() {
    print("Employee ID: $id");
    print("Employee Name: $name");
  }
}
```

Porém, em Dart, é possível criar Named Constructors, nesse caso permitindo que a classe tenha múltiplos construtores.

```
const int idOrigin = 0;
const String nameOrigin = '';

class Employee {
  int id;
  String name;

  Employee(this.id, this.name);

  Employee.origin()
      : id = idOrigin,
        name = nameOrigin;

  void displayEmployeeDetails() {
    print("Employee ID: $id");
    print("Employee Name: $name");
  }
}
```

É importante destacar também que uma classe-filha não irá herdar o construtor de uma classe-mãe, se o construtor não for definido na classe-filha, ele receberá o construtor padrão (sem nome e argumento).

MODIFICADORES DE CLASSE

Utéis para definir comportamentos de classes.

ABSTRACT

Uma classe abstrata como o nome diz serve de abstração para demais classes. Ela não será utilizada, servirá somente de molde.

```
abstract class Vehicle {
  void moveForward(int meters);
}
```

```
// Error: Can't be constructed.
Vehicle myVehicle = Vehicle();

// Can be extended.
class Car extends Vehicle {
  int passengers = 4;
  // ...
```

```
}

// Can be implemented.
class MockVehicle implements Vehicle {
  @override
  void moveForward(int meters) {
    // ...
  }
}
```

No exemplo podemos verificar que a classe não pode ser instanciada diretamente, portanto ela terá que ser estendida por meio de outra classe ou implementada.

EXTENDS x IMPLEMENTS

O que seria a implementação de uma classe? Qual a diferença entre implementação e extensão?

A implementação de uma classe permite que ela possa definir implementações próprias de um conjunto de métodos da classe que ela está implementando.

```
// A person. The implicit interface contains greet().
class Person {
  // In the interface, but visible only in this library.
  final String _name;

  // Not in the interface, since this is a constructor.
  Person(this._name);

  // In the interface.
  String greet(String who) => 'Hello, $who. I am $_name.';
}

// An implementation of the Person interface.
class Impostor implements Person {
  String get _name => '';

  String greet(String who) => 'Hi $who. Do you know who I am?';
}

String greetBob(Person person) => person.greet('Bob');

void main() {
  print(greetBob(Person('Kathy')));
  print(greetBob(Impostor()));
}
```

Para todos os métodos da classe Person, a classe Impostor precisa definir suas próprias implementações. Ou seja, a classe Impostor utiliza da interface da classe Person para criar sua própria assinatura se baseando em uma estrutura pré-definida. Diferentemente do identificador @override da extensão de uma classe. No exemplo anterior, ao utilizar o @override no extends era possível realizar a substituição de um método da classe-mãe por um método da classe-filha, se baseando na mesma estrutura pré-definida.

Ou seja, implements é utilizado quando uma classe deseja aderir a interface de outra classe, implementando suas próprias especificações. Já extends é utilizado quando uma classe necessita herdar propriedades e comportamentos, podendo modificá-los ou adicionar novos.

BASE

Uma classe base não pode ser implementada fora do seu escopo (biblioteca). As classes que irão utilizar a classe base precisam ser identificadas com base, final ou sealed.

```
base class Vehicle {
  void moveForward(int meters) {
    // ...
  }
}
```

```
// Can be constructed.
Vehicle myVehicle = Vehicle();

// Can be extended.
base class Car extends Vehicle {
  int passengers = 4;
  // ...
}

// ERROR: Can't be implemented.
base class MockVehicle implements Vehicle {
  @override
  void moveForward() {
    // ...
  }
}
```

FINAL

Uma classe final não permite a implementação ou extensão fora de seu escopo.

```
final class Vehicle {
  void moveForward(int meters) {
    // ...
  }
}
```

```
// Can be constructed.
Vehicle myVehicle = Vehicle();

// ERROR: Can't be inherited.
class Car extends Vehicle {
  int passengers = 4;
  // ...
}
```

```
class MockVehicle implements Vehicle {
  // ERROR: Can't be implemented.
  @override
  void moveForward(int meters) {
    // ...
  }
}
```

INTERFACE

Uma classe interface define que ela servirá somente como interface, não podendo ser estendida. É comum definir a classe como abstract interface.

```
interface class Vehicle {
  void moveForward(int meters) {
    // ...
  }
}
```

```
// Can be constructed.
Vehicle myVehicle = Vehicle();

// ERROR: Can't be inherited.
class Car extends Vehicle {
  int passengers = 4;
  // ...
}

// Can be implemented.
class MockVehicle implements Vehicle {
  @override
  void moveForward(int meters) {
    // ...
  }
}
```

SEALED

Uma classe sealed não permite a implementação ou extensão fora de seu escopo. Implicitamente abstract, porém as classes-filha não.

Não possui construtor próprio portanto.

```
sealed class Vehicle {}

class Car extends Vehicle {}

class Truck implements Vehicle {}

class Bicycle extends Vehicle {}

// ERROR: Can't be instantiated.
Vehicle myVehicle = Vehicle();

// Subclasses can be instantiated.
Vehicle myCar = Car();

String getVehicleSound(Vehicle vehicle) {
  // ERROR: The switch is missing the Bicycle subtype or a default case.
  return switch (vehicle) {
    Car() => 'vroom',
    Truck() => 'VROOOOMM',
  };
}
```

A classe sealed reconhece todos os subtipos, e assim é possível realizar um switch sobre eles.

MIXIN

Um mixin é um código reutilizável. Um conceito que pode ser visto desde transpiladores de css a template renders.

É possível declarar ambos classe e mixin utilizando mixin class.

```
mixin Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
      print('Waving hands');
    } else {
      print('Humming to self');
    }
  }
}
```

```
class Musician extends Performer with Musical {
  // ...
}

class Maestro extends Person with Musical, Aggressive, Dement
  Maestro(String maestroName) {
    name = maestroName;
    canConduct = true;
  }
}
```

No caso da classe Maestro, ela consegue fazendo da palavra with utilizar os diversos mixins Musical, Aggressive e Demented.

Se for necessário restringir o uso de um mixin, é possível o fazer com a palavra-chave on.

```
class Musician {
  // ...
```

```
}

mixin MusicalPerformer on Musician {
  // ...
}

class SingerDancer extends Musician with MusicalPerformer {
  // ...
}
```

O exemplo nos indica que somente classes que estendem ou implementam a classe Musician podem usar o mixin MusicalPerformer.

REFERÊNCIAS

https://flutterwithjanith.medium.com/dart-oop-object-oriented-programming-concepts-7b10102fa45e

https://dart.dev/language/classes