Les classes

Temps de lecture : 10 minutes

Généralités sur les classes en `Dart`

En `Dart` tous les objets sont les instances d'une classe, et toutes les classes descendent de la classe `Object`.

En `Dart`, toutes les classes ont exactement une classe parente, appelée `superclass`, sauf la classe `Object`.

Créer une classe

Pour créer une classe il suffit de faire :

```
class Point {
}
```

Pour obtenir une instance d'une classe il suffit de faire :

```
var instance1 = new Point();
// ou car new est optionnel en Dart :
var instance2 = Point();
```

Le contenu des classes

Les classes peuvent contenir des fonctions, appelées **méthodes**, et des données, appelées **variables d'instance**.

Nous allons commencer par vois les variables d'instance, puis nous verrons les méthodes après avoir étudié les constructeurs.

Les variables d'instance

Pour définir des variables d'instance il suffit de faire :

```
class Point {
  // Déclaration d'une variable d'instance x de type num et initialisati
on à null :
  num? x;
  // Déclaration d'une variable d'instance y de type num et initialisati
on à 0 :
  num y = 0;
}
```

Notez le point d'interrogation après le type `num`. Il signifie que la variable peut être nulle.

Depuis une version récente de Dart, a été introduit la "null safety", à savoir qu'il faut explicitement déclarer qu'une variable ou un argument peut être nul en ajoutant un point d'interrogation. C'est une vérification supplémentaire introduite par Dart.

Pour invoquer une méthode ou utiliser une propriété sur l'instance d'une classe il faut utiliser `.methode()` ou `.propriete`.

Les constructeurs

En `Dart` les constructeurs de base ont le même nom que la classe auxquels ils appartiennent et ce sont de simples fonctions :

```
class Point {
  // raccourci syntaxique pour déclarer deux variables du même type :
  num? x, y;

  Point(num x, num y) {
    this.x = x;
    this.y = y;
  }
}
```

Ici, `this` se réfère à l'instance courante. En `Dart`, nous n'utilisons `this` que dans le cas d'un conflit de nom.

En effet, ici la classe et le constructeur ont le même temps, et c'est pour cette raison que nous utilisons `this` pour se référer à l'instance courante.

Il existe un raccourci syntaxique pour écrire exactement le même constructeur :

```
class Point {
  num? x, y;

  Point(this.x, this.y);
}
```

Les constructeurs nommés

Les constructeurs nommés, ou `named constructor` permettent d'avoir plusieurs constructeurs **dans une même classe** !

Pour créer d'autres constructeurs, il suffit d'utiliser `NomClasse.NomConstructeur()`, comme ici :

```
class Point {
  num? x, y;

  Point(this.x, this.y);

  Point.origin() {
    x = 0;
    y = 0;
  }
}
```

Les méthodes d'instance

Les méthodes d'instance sont des fonctions définies sur une classe et accessibles sur les instances de cette classe.

Voici un exemple de méthode d'instance :
https://dartpad.dev/embed-inline.html

Les méthodes et les propriétés statiques

Par opposition aux méthodes d'instance et aux propriété d'instance, nous pouvons définir des propriétés et méthodes qui ne seront accessibles que depuis l'objet de la classe.

Elles ne seront pas accessibles sur les instances de la classe.

Pour ce faire, il suffit d'utiliser `static` :

```
class Homme {
  String? prenom, nom;
  static const genre = 'masculin';

  Homme(this.prenom, this.nom);
  static direBonjour() => print('Bonjour !');
}

void main() {
  Homme jean = Homme('Jean', 'Dupont');
  print(Homme.genre); // masculin
  Homme.direBonjour(); // Bonjour !
  print(jean.genre); // Error: The getter 'genre' isn't defined for the
class 'Homme'.
}
```

Les interfaces

Une interface définit la syntaxe à laquelle une classe doit adhérer.

Les interfaces définissent un ensemble de méthodes disponibles sur un objet.

Les interfaces implicites

En `Dart`, il n'y a pas de déclaration d'`interface`, en effet chaque classe définit implicitement une `interface`.

Cette interface contient toutes les propriétés et les méthodes implémentées sur la classe, nous pouvons le voir comme le "type" d'une classe.

Elle comprend également toutes les interfaces implémentées par la classe.

L'implémentation d'interface

Lorsqu'une classe implémente une autre classe, cela signifie qu'elle déclare toutes les propriétés définies sur la première classe.

Par exemple :

```
// Prenons une classe :
class ClasseA {
  String? prop1;

  ClasseA(this.prop1);

  String methode1(String param) => 'Un $param.';
}

// La classe A a donc comme interface implicite, une propriété d'instanc
e
// de type String, et une méthode qui prend en paramètre une String.

class ClasseB implements ClasseA {
  String prop1 = 'une valeur';
  String? prop2;

  String methode1(String param) => 'Retourne autre chose';
}
```

La `ClasseB` implémente la `ClasseA` car elle respecte son `API`. Elle doit contenir toutes les propriétés et méthodes de la `ClasseA` mais peut en implémenter d'autres, comme par exemple `prop2`.

Il faut également respecter les types des propriétés, les types du retour et des paramètres de la `ClasseA`.

Nous reverrons des cas concrets plus tard.

Les classes abstraites

Parfois nous voulons définir des interfaces sans les implémenter directement.

Dans ce cas, en `Dart`, il faut utiliser une **classe abstraite**. Il s'agit en fait d'une classe dont l'interface implicite ne sera implémentée que dans une autre classe.

Les classes abstraites ne peuvent pas être instanciée.

Elles peuvent être implémentées par d'autres classes.

```
abstract class UneClasseAbstraite {
  String proprieteAbstraite;
```

```
  void methodeAbstraite();
}

class UneClasseImplementee implements UneClasseAbstraite {
  String proprieteAbstraite = 'valeur';

  void methodeAbstraite() => print(proprieteAbstraite);
}
```

Nous appelons méthodes abstraites les méthodes définies sur une classe abstraite car elles ne sont pas implémentés : elles ne font rien.

Les `getters` et les `setters`

Les `getters` et les `setters` sont des méthodes spéciales qui permettent de définir comment des propriétés d'instance sont lus et / ou écrites.

Par défaut les propriétés d'instance ont un `getter` et un `setter` implicites :
https://dartpad.dev/embed-inline.html

Mais il est possible de définir ses propres `getters` et `setters` également :
https://dartpad.dev/embed-inline.html

L'héritage

Contrairement au `JavaScript`, nous ne sommes pas dans un langage prototypal. Les classes ne sont pas du sucre syntaxique.

L'héritage est la possibilité de créer des classes à partir d'autres classes.

Nous appelons la classe à partir duquelle une autre classe est créée **la classe parente ou la** `super class`.

La classe créée à partir de la classe parente s'appelle **la classe fille ou la** `sub class`.

Il existe deux types d'héritage : l'héritage simple et l'héritage multiple.

L'héritage multiple est le fait qu'une classe peut hériter de plusieurs classes, ce n'est pas possible de `Dart` qui ne supporte que l'héritage simple.

En revanche `Dart` supporte l'héritage **multi-niveaux** : une classe peut hériter d'une classe, héritant elle-même d'une autre classe.

Utilisation d'`extends`

Une classe hérite d'une autre classe en utilisant `extends` .

La classe fille hérite de toutes les propriétés et les méthodes de la classe parente, mais pas de son constructeur.

```
void main() {
    Enfant enfant = new Enfant();
    enfant.direBonjour();
}
class Parent {
    void direBonjour() {
      print("Bonjour !!");
    }
}
class Enfant extends Parent {}
```

Utilisation de `super`

Le mot clé `super` permet de se référer à la classe parente :
https://dartpad.dev/embed-inline.html

Il est possible d'utiliser le constructeur de la classe parente dans la classe fille en utilisant `: super` :
https://dartpad.dev/embed-inline.html

Utilisation de `@override`

Si dans une classe fille vous voulez remplacer une méthode d'instance, un `getter` ou un `setter` , vous pouvez utiliser l'annotation `@override` :

```
class ClasseParente {
  void direBonjour() => print('Bonjour du parent !');
}

class ClasseEnfant extends ClasseParente {
  @override
```

```
  void direBonjour() => print('Bonjour de l'enfant !');
}

void main() {
  ClasseEnfant enfant = ClasseEnfant();
  enfant.direBonjour(); // Bonjour de l'enfant !
}
```

Les types énumérés

Les types émunérés, souvent appelés `enum` sont un type spécifique de classe `Dart` qui permettent de représenter un nombre fixe de valeurs constantes.

```
enum Direction { gauche, droite, devant, derriere }
```

Il est possible d'accéder à toutes les valeurs d'un `enum` avec la propriété `values` :

```
List<Direction> directions = Direction.values;
```

Vous pouvez vous servir d'un `enum` pour les `switch / case` par exemple, vous aurez une erreur si vous oubliez un cas :
https://dartpad.dev/embed-inline.html

Les génériques

Les `generics` permettent de spécifier qu'une collection en `Dart` ne peut contenir que des valeurs d'un certain type.

Ils fonctionnent sur toutes les collections `Dart`. Pour le moment, nous avons vu deux types de collections : les `lists` et les `maps`.

La syntaxe est la suivante :

```
Collection <type> identifiant = assignation
```

Nous en avons déjà vus dans le chapitre précédemment, voici un exemple :

```
List<String> noms = List<String>();
noms.addAll(['Jean', 'Paul']);
```

```
noms.add(42); // Error: The argument type 'int' can't be assigned to
// the parameter type 'String'.
```

Les collections qui utilisent des génériques sont appelés des **collections** `type-safe`.

Il est également possible d'utiliser un type indéfini souvent indiqué par `T` pour `type`, par convention.

Nous pouvons également typer des fonctions et des méthodes avec un type générique en faisant :

```
function<T> () { … };
```

Avec cette notation nous passons en paramètre un type générique que nous utiliserons dans la fonction ou la méthode.

Prenons un exemple :
https://dartpad.dev/embed-inline.html

Dans cette exemple nous définissons une fonction `soustraire()` et nous la typons en indiquant qu'elle va retourner un `type T`, nous restreignons également les arguments qu'elle peut recevoir grâce à `extends`.

En effet, `<T extends num>` signifie que `T` ne peut être qu'un `num` donc un `double` ou un `int`.

Ensuite, nous typons également ses paramètres avec `T` et `List<T>`.

Nous pouvons ainsi utiliser la même fonction pour des `doubles` et des `ints`, sans perdre les bénéfices du typage !