Polymorphisme dans Dart

Objectifs du cours

Dans ce cours, vous découvrirez le polymorphisme dans le langage de programmation Dart à l'aide d'exemples. Avant d'en apprendre davantage sur le polymorphisme dans Dart, vous devez avoir une compréhension de base d'héritage à Dart.

1. ## Comprendre le concept de polymorphisme en Dart.

   - **`Poly`** signifie **plusieurs** et **`morph`** signifie **formes**.
   - Le polymorphisme est la capacité d'un objet à prendre plusieurs formes. En tant qu'humains, nous avons la capacité de prendre de nombreuses formes. Nous pouvons être un étudiant, un enseignant, un parent, un ami, etc.
   - De même, en programmation orientée objet, le polymorphisme est la capacité d'un objet à prendre plusieurs formes.
   - Le polymorphisme est un concept clé de la programmation orientée objet (POO) qui permet à un objet de prendre différentes formes ou comportements en fonction du contexte dans lequel il est utilisé. En Dart, le polymorphisme est réalisé grâce à l'héritage et à la substitution de méthodes.
   - En Dart, le polymorphisme se réfère à la capacité d'un objet à prendre différentes formes. Cela peut se manifester à travers l'héritage et les interfaces.
   - Le `polymorphisme` permet d'utiliser une interface unique pour des types de données différents. Imaginez un bouton "Lecture" : il peut lancer une musique, une vidéo ou un podcast. L'action est la même (Play), mais le comportement interne diffère selon l'objet.
   - Pour mieux comprendre le polymorphisme dans Dart, examinons un exemple concret. Supposons que nous ayons une classe de base appelée Animal :

```
class Animal {
  void faireDuBruit() {
    print('L'animal fait du bruit !');
  }
}
```

   - Maintenant, nous allons créer deux sous-classes de Animal, à savoir Chien et Chat. Ces sous-classes hériteront de la classe Animal et pourront également redéfinir (ou substituer) la méthode faireDuBruit() selon leurs propres comportements :

apcpedagogie.com

```
class Chien extends Animal {
  @override
  void faireDuBruit() {
    print('Le chien aboie !');
  }
}

class Chat extends Animal {
  @override
  void faireDuBruit() {
    print('Le chat miaule !');
  }
}
```

- Maintenant, nous pouvons créer des instances de Chien et Chat et les utiliser de manière polymorphique en tant qu'objets de type Animal :

```
void main() {
  Animal animal = Animal();
  Animal chien = Chien();
  Animal chat = Chat();

  animal.faireDuBruit(); // Affiche : L'animal fait du bruit !
  chien.faireDuBruit(); // Affiche : Le chien aboie !
  chat.faireDuBruit(); // Affiche : Le chat miaule !
}
```

- Dans cet exemple, bien que les variables chien et chat soient déclarées comme des objets de type Animal, elles conservent leurs comportements spécifiques définis dans les sous-classes Chien et Chat. C'est ce qu'on appelle le polymorphisme, où un objet de type plus général peut se comporter de manière spécifique en fonction de sa sous-classe réelle.
- Le polymorphisme est utile car il permet de créer des interfaces génériques et flexibles, où différentes classes peuvent être utilisées de manière interchangeable, tant qu'elles héritent de la même classe de base ou implémentent la même interface.
- En résumé, le polymorphisme dans Dart est réalisé grâce à l'héritage et à la substitution de méthodes.
- Pour bien comprendre comment le polymorphisme structure un projet, séparons notre code en plusieurs fichiers. Cet exemple simule un système de traitement de paiements (très courant en Flutter).

apcpedagogie.com

- **1. Le contrat : payment_method.dart**
  - Ici, on définit une classe abstraite. Elle sert de "moule". On ne peut pas créer un objet PaymentMethod directement, mais on garantit que n'importe quel moyen de paiement aura une méthode pay.

```
// payment_method.dart
abstract class PaymentMethod {
/*le mot-clé abstract définit une classe abstraite, c'est-à-dire une classe qui ne peut pas être instanciée directement. Elle sert de modèle ou de contrat pour les classes qui en héritent.*/
  final double amount;

  PaymentMethod(this.amount);

  // Signature de la méthode sans implémentation
  void pay();
}
```

- **2. Les implémentations : credit_card.dart et paypal.dart**
  - Chaque classe va hériter de la base et utiliser @override pour définir sa propre logique.

```
// credit_card.dart
import 'payment_method.dart';

class CreditCard extends PaymentMethod {
  CreditCard(double amount) : super(amount);

  @override
  void pay() {
    print("Paiement de $amount€ effectué par Carte Bancaire (chiffrement SSL).");
  }
}
```

```
// paypal.dart
import 'payment_method.dart';
```

apcpedagogie.com

```
class PayPal extends PaymentMethod {
  PayPal(double amount) : super(amount);

  @override
  void pay() {
    print("Redirection vers PayPal... Paiement de $amount€ validé !");
  }
}
```

- **3. Le chef d'orchestre : main.dart**
  - C'est ici que la magie du polymorphisme opère. On manipule des objets différents comme s'ils étaient du même type (PaymentMethod).

```
import 'payment_method.dart';
import 'credit_card.dart';
import 'paypal.dart';

void main() {
  // Liste polymorphique : elle accepte n'importe quoi qui "EST UN"
PaymentMethod
  List panier = [
    CreditCard(45.50),
    PayPal(12.00),
    CreditCard(100.00),
  ];

  print("--- Début du traitement des paiements ---");

  // On traite chaque élément sans savoir si c'est une carte ou PayPal
  for (var methode in panier) {
    methode.pay(); // Le programme choisit la bonne version au moment de
l'exécution
  }
}
```

- **Ce qu'il faut retenir de cet exemple :**
- **Flexibilité**: Si demain vous voulez ajouter le paiement en Bitcoin, vous créez juste un fichier bitcoin.dart qui extends PaymentMethod. Vous n'avez aucun besoin de modifier la boucle for dans votre main.dart.
- **Sécurité**: Grâce à @override dans les fichiers credit_card et paypal, vous êtes certain de respecter le nom de la méthode imposé par le parent.

apcpedagogie.com

- **Abstraction** : Le main se fiche de savoir comment on paye, il veut juste savoir que l'objet peut payer.

2. Le concept de classe abstraite en Dart

1. Qu'est-ce qu'une classe abstraite ?

- **Une classe abstraite** est une classe qui ne peut pas être instanciée. On ne peut pas créer un objet directement à partir d'elle (ex: Personne() n'a pas de sens, car on est soit un Étudiant, soit un Enseignant).
  - **Mot-clé**: `abstract`
  - **But**: Définir un modèle commun et forcer les sous-classes à implémenter certains comportements.

2. Anatomie d'une classe abstraite

- Une classe abstraite peut contenir deux types de choses :
  - **Des méthodes concrètes**: Avec du code (logique partagée par tous).
  - **Des méthodes abstraites**: Sans corps (seulement la signature), que les sous-classes doivent obligatoirement écrire.
- **Exemple** : Le système éducatif

```
// personne.dart
abstract class Personne {
  String nom;

  Personne(this.nom);

  // MÉTHODE ABSTRAITE : Pas de corps { }, se termine par un ";"
  // On force chaque sous-classe à définir comment elle se présente.
  void sePresenter();

  // MÉTHODE CONCRÈTE : Code partagé par tout le monde
  void manger() {
    print("$nom est en train de manger à la cantine.");
  }
}
```

3. L'implémentation dans les sous-classes

apcpedagogie.com

- Quand une classe hérite (extends) d'une classe abstraite, elle a une obligation contractuelle de remplir les "blancs" (les méthodes abstraites).
- **L'analogie du Formulaire d'Inscription**
- Imagine que l'administration de l'université te donne un formulaire (la classe abstract Personne).
  - Sur ce formulaire, il y a une case obligatoire : "Décrivez votre activité principale".
  - Si tu rends le formulaire sans remplir la case, l'administration le rejette immédiatement. C'est l'erreur de compilation.
  - L'obligation, c'est l'assurance pour l'université que chaque personne enregistrée a une activité, même si elle est différente pour chacun.

```
// etudiant.dart
class Etudiant extends Personne {
  int niveau;

  Etudiant(String nom, this.niveau) : super(nom);

  @override
  void sePresenter() {
    print("Je suis l'étudiant $nom, en classe de $niveau.");
  }
}

// enseignant.dart
class Enseignant extends Personne {
  String matiere;

  Enseignant(String nom, this.matiere) : super(nom);

  @override
  void sePresenter() {
    print("Je suis M. $nom, professeur de $matiere.");
  }
}
```

4. **Exemple : Les animaux (niveau débutant)**

   - **Étape 1 : La classe abstraite Animal**

apcpedagogie.com

```
// animal.dart
abstract class Animal {
  final String nom;
  final int age;

  Animal(this.nom, this.age);

  // ? Méthode abstraite : OBLIGATOIRE à implémenter
  void crier();        // Comment cet animal fait-il du bruit ?
  void seDeplacer();   // Comment cet animal se déplace-t-il ?

  // ? Méthode concrète : déjà implémentée, réutilisable
  void presenter() {
    print("Je suis $nom, j'ai $age ans.");
  }

  // ? Méthode avec implémentation partielle
  void dormir() {
    print("$nom dort paisiblement... ?");
  }
}
```

- **Étape 2 : Les classes concrètes**

```
// chien.dart
class Chien extends Animal {
  Chien(String nom, int age) : super(nom, age);

  @override
  void crier() {
    print("$nom dit : Wouf ! Wouf ! ?");
  }

  @override
  void seDeplacer() {
    print("$nom court sur 4 pattes.");
  }
}

// oiseau.dart
class Oiseau extends Animal {
```

apcpedagogie.com

```
  Oiseau(String nom, int age) : super(nom, age);

  @override
  void crier() {
    print("$nom dit : Cuicui ! ?");
  }

  @override
  void seDeplacer() {
    print("$nom vole dans le ciel ✈");
  }
}

// poisson.dart
class Poisson extends Animal {
  Poisson(String nom, int age) : super(nom, age);

  @override
  void crier() {
    print("$nom fait des bulles... ?"); // Les poissons ne crient pas
vraiment !
  }

  @override
  void seDeplacer() {
    print("$nom nage dans l'eau ?");
  }
}
```

- **Étape 3 : Utilisation**

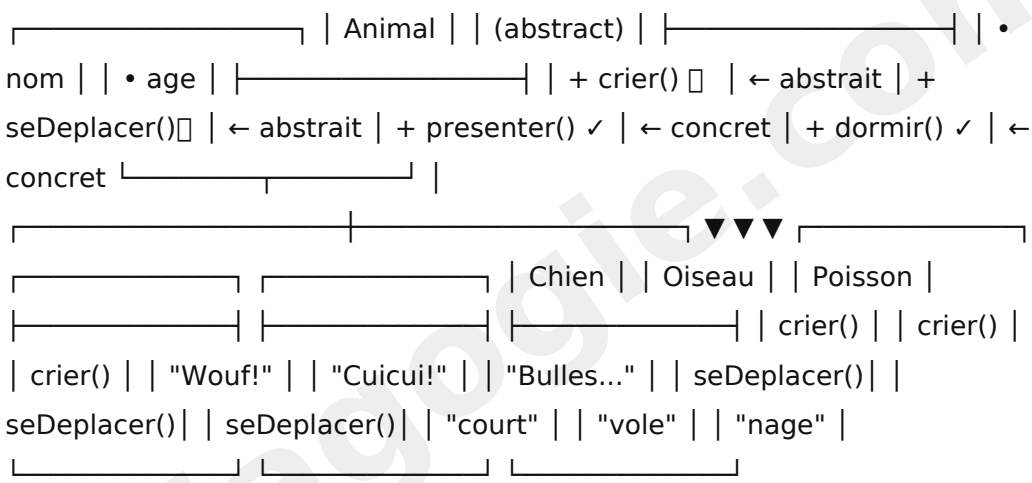
```
void main() {
  // ❌ Ceci ne fonctionne PAS :
  // Animal generic = Animal("Generic", 3);
  // → Erreur : on ne peut pas instancier une classe abstraite

  // ✅ On utilise les classes concrètes :
  List animaux = [
    Chien("Rex", 3),
    Oiseau("Tweety", 1),
    Poisson("Nemo", 2),
  ];
```

apcpedagogie.com

```
  for (var animal in animaux) {
    animal.presenter();      // Méthode commune héritée
    animal.crier();          // Méthode spécifique à chaque animal
    animal.seDeplacer();     // Méthode spécifique à chaque animal
    animal.dormir();         // Méthode commune avec implémentation
    print("---");
  }
}
```

- **Schéma visuel du concept**



- **La différence entre "vide" et "abstraite"**

Type de méthode	Syntaxe	Corps	Obligatoire à implémenter ?
Méthode abstraite	void pay();	❌ Aucun (juste ;)	✅ OUI, obligatoire
Méthode concrète vide	void pay() {}	✅ Présent (même vide)	❌ Non, optionnel

3. Quel est le but de `@override` dans Flutter (Dart) ?

- Dans Flutter, l'annotation `@override` est utilisée pour indiquer qu'une méthode d'une sous-classe est destinée à remplacer une méthode du même nom dans sa superclasse ou une interface. C'est une manière d'indiquer explicitement que la méthode fournit intentionnellement une nouvelle implémentation ou un nouveau comportement pour la méthode superclasse/interface.

apcpedagogie.com

- Lorsque vous marquez une méthode avec `@override`, cela permet de garantir que vous remplacez réellement une méthode et que vous ne créez pas accidentellement une nouvelle méthode avec un nom similaire. S'il n'y a pas de méthode correspondante dans la superclasse ou l'interface, l'analyseur Dart générera une erreur.
- Voici un exemple pour illustrer l'utilisation de `@override` dans Flutter :

```
class Animal {
  void makeSound() {
    print('Unknown sound');
  }
}

class Cat extends Animal {
  @override
  void makeSound() {
    print('Meow!');
  }
}

void main() {
  Cat cat = Cat();
  cat.makeSound(); // Output: Meow!
}
```

- Dans cet exemple, nous avons une classe **Animal** avec une méthode **makeSound()**. La classe **Cat** étend **Animal** et remplace la méthode **makeSound()** à l'aide de l'annotation `@override`. Lors de l'appel de **makeSound()** sur une instance de `Cat`, la méthode remplacée dans **Cat** est exécutée, produisant le résultat "Meow !" au lieu du "**Son inconnu**" par défaut défini dans **Animal**.
- L'utilisation de `@override` est une bonne pratique car elle améliore la lisibilité du code et permet d'éviter les erreurs accidentelles lors du sous-classement ou de l'implémentation d'interfaces dans Flutter.

4. Remplacement de méthode dans Dart

- Le remplacement de méthode se produit dans Dart lorsqu'une classe enfant tente de remplacer la méthode de la classe parent. Lorsqu'une classe enfant étend une classe parent, elle obtient un accès complet aux méthodes de la classe parent et remplace ainsi les méthodes de la classe parent. Ceci est réalisé en redéfinissant la même méthode présente dans la classe parent.

apcpedagogie.com

- Le rôle principal des remplacements dans Dart est d'augmenter la réutilisabilité du code et de fournir une extensibilité.
- En personnalisant les méthodes définies dans une classe parent dans une sous-classe, vous pouvez disposer des mêmes fonctionnalités de base mais vous comporter différemment dans des contextes différents.
- Cette méthode est utile lorsque vous devez exécuter différentes fonctions pour une classe enfant différente, nous pouvons donc simplement redéfinir le contenu en le remplaçant.

> 1. *Une méthode ne peut être substituée que dans la classe enfant, pas dans la classe parent elle-même.*
> 2. *Les méthodes définies dans la classe enfant et la classe parent doivent être la copie exacte, du nom à la liste da'rguments, à l'exception du contenu présent à l'intérieur de la méthode, c'est-à-dire qu'il peut et ne peut pas être le même.*
> 3. *Une méthode déclarée `final` ou `static` dans la classe parent ne peut pas être remplacée par la classe enfant.*
> 4. *Les constructeurs de la classe parent ne peuvent pas être hérités et ne peuvent donc pas être remplacés par la classe enfant.*

1. **Surcharge de Méthode en Dart : Surcharge de Méthode en Dart**

   - La surcharge de méthode (method overriding) est une technique qui vous permet de créer une méthode dans la classe fille avec le même nom que la méthode dans la classe parente. Ainsi, la méthode dans la classe fille remplace la méthode dans la classe parente.

```
class ParentClass {
  void functionName() {
    //....
  }
}
class ChildClass extends ParentClass {
  @override
  void functionName() {
    // ....
  }
}
```

apcpedagogie.com

2. **Le Polymorphisme en Dart : Substitution de Méthodes et Héritage de Classes**

```
class Animal {
  // Fonction pour l'action de manger
  void eat() {
    print("L'animal mange");
  }
}

class Dog extends Animal {
  // Remplace la fonction eat() de la classe parente
  @override
  void eat() {
    print("Le chien mange");
  }
}

void main() {
  // Création d'une instance de la classe Animal
  Animal animal = Animal();
  // Appel de la fonction eat() de la classe Animal
  animal.eat();

  // Création d'une instance de la classe Dog
  Dog dog = Dog();
  // Appel de la fonction eat() de la classe Dog, qui a été substituée
  dog.eat();
}
```

- Dans cet exemple, nous avons une classe **Animal** avec une méthode **eat()**. Cette méthode est ensuite remplacée dans la classe fille **Dog** avec une nouvelle implémentation. Lorsque nous appelons la méthode **eat()** sur une instance de **Animal**, nous obtenons le message "**L'animal mange**", tandis que lorsqu'elle est appelée sur une instance de **Dog**, nous obtenons le message "**Le chien mange**". Cela montre comment la substitution de méthodes fonctionne en Dart.

3. **Le Polymorphisme en Dart : Surcharge de Méthode en Dart**

apcpedagogie.com

5. Exemples

```
class Graphique {
  late int x; // Coordonnée x du centre de l'objet
  late int y; // Coordonnée y du centre de l'objet

  Graphique(int x, int y) {
    this.x = x; // Assigner la coordonnée x passée en paramètre à la variable d'instance x
    this.y = y; // Assigner la coordonnée y passée en paramètre à la variable d'instance y
  }

  void identifie() {
    print("Je suis une forme géométrique"); // Afficher le message "Je suis une forme géométrique"
  }

  void affiche() {
    identifie(); // Appeler la méthode identifie() pour afficher le type de forme géométrique
    print("Le centre de l'objet se trouve dans : $x et $y"); // Afficher les coordonnées du centre de l'objet
  }

  double surface() {
    return 0; // Retourner une surface nulle (0) par défaut
  }
}

class Cercle extends Graphique {
  late double rayon; // Rayon du cercle

  Cercle(int x, int y, double r) : super(x, y) {
    rayon = r; // Assigner le rayon passé en paramètre à la variable d'instance rayon
  }

  @override
  void identifie() {
    print("Je suis un cercle"); // Afficher le message "Je suis un cercle"
```

apcpedagogie.com

```
  }

  @override
  double surface() {
    return rayon * 2 * 3.14; // Calculer et retourner la surface du cercle en
utilisant la formule : rayon * 2 * π (approximation de π avec la valeur 3.14)
  }
}

class Rectangle extends Graphique {
  late int longueur; // Longueur du rectangle
  late int largeur; // Largeur du rectangle

  Rectangle(int x, int y, int l1, int l2) : super(x, y) {
    longueur = l1; // Assigner la longueur passée en paramètre à la variable
d'instance longueur
    largeur = l2; // Assigner la largeur passée en paramètre à la variable
d'instance largeur
  }

  @override
  double surface() {
    return longueur * largeur.toDouble(); // Calculer et retourner la surface du
rectangle en multipliant la longueur par la largeur
  }

  @override
  void identifie() {
    print("Je suis un rectangle"); // Afficher le message "Je suis un rectangle"
  }
}

void main() {
  Graphique g = Graphique(3, 7); // Créer une instance de la classe Graphique avec
les coordonnées (3, 7)
  g.identifie(); // Appeler la méthode identifie() de l'objet g pour afficher le
type de forme géométrique

  g = Cercle(4, 8, 10); // Créer une instance de la classe Cercle avec les
coordonnées (4, 8) et un rayon de 10
  g.identifie(); // Appeler la méthode identifie() de l'objet g pour afficher le
type de forme géométrique
```

apcpedagogie.com

```
  g = Rectangle(7, 9, 10, 3); // Créer une instance de la classe Rectangle avec les coordonnées (7, 9), une longueur de 10 et une largeur de 3
  g.identifie(); // Appeler la méthode identifie() de l'objet g pour afficher le type de forme géométrique
}
```

- **Interprétation**

  - Dans le code Dart que vous avez fourni, la variable g est d'abord déclarée comme étant de type Graphique, puis son référentiel est modifié pour référencer successivement des instances des classes dérivées Cercle et Rectangle. Cette flexibilité est permise grâce à la compatibilité entre un type de classe de base et une référence de classe dérivée.
  - Lorsque vous appelez la méthode identifie() sur la variable g, la liaison dynamique des méthodes est utilisée. Cela signifie que la méthode appelée dépend du type réel de l'objet auquel g fait référence à ce moment-là. Ainsi, lorsque g fait référence à une instance de la classe Graphique, la méthode identifie() de la classe Graphique est appelée. Lorsque g fait référence à une instance de la classe Cercle, la méthode identifie() de la classe Cercle est appelée, et de même pour la classe Rectangle.
  - Cela démontre le polymorphisme, qui est l'un des concepts clés de la programmation orientée objet. Le polymorphisme permet à des objets de différentes classes de répondre de manière spécifique aux mêmes appels de méthode, en fonction de leur type réel.
  - En résumé, grâce à l'utilisation des classes de base et dérivées, ainsi que la liaison dynamique des méthodes, vous pouvez manipuler des objets de différentes classes de manière polymorphique et obtenir un comportement spécifique à chaque classe lors de l'appel des méthodes.

6. Application

1. **Énoncé**

   - **Développer les classes suivantes :**
     - **Entité** : comportant les champs privés nom, prénom et date de naissance, un constructeur pour initialiser les données, et une méthode polymorphe Afficher pour afficher les données de chaque entité.

apcpedagogie.com

- **Travailleur**: étendant Entité, avec en plus un champ Salaire accompagné de sa propriété, un constructeur et la redéfinition de la méthode Afficher.
- **Manager**: étendant Travailleur, avec en plus un champ Service accompagné de sa propriété, un constructeur et la redéfinition de la méthode Afficher.
- **PDG**: étendant Manager, avec en plus un champ Société accompagné de sa propriété, un constructeur et la redéfinition de la méthode Afficher.

- **Travail à accomplir:**
  - Composer les classes **Entité**, **Travailleur**, **Manager** et **PDG**.
  - Élaborer un programme de test comportant un tableau de huit entités : cinq travailleurs, deux managers et un PDG (8 références de la classe Entité contenant 5 instances de Travailleur, 2 de Manager et 1 de PDG).
  - Présenter l'ensemble des éléments du tableau en utilisant une boucle for.
  - Présenter l'ensemble des éléments du tableau en utilisant une boucle foreach.

2. **Solution**

Code Dart

```
class Entite {
  String nom;
  String prenom;
  DateTime dateNaissance;

  Entite(this.nom, this.prenom, this.dateNaissance);

  void afficher() {
    print('Nom: $nom, Prénom: $prenom, Date de naissance: $dateNaissance');
  }
}

class Travailleur extends Entite {
  double salaire;

  Travailleur(String nom, String prenom, DateTime dateNaissance, this.salaire)
      : super(nom, prenom, dateNaissance);

  @override
  void afficher() {
    super.afficher();
```

apcpedagogie.com

```
    print('Salaire: $salaire');
  }
}

class Manager extends Travailleur {
  String service;

  Manager(String nom, String prenom, DateTime dateNaissance, double salaire,
this.service)
      : super(nom, prenom, dateNaissance, salaire);

  @override
  void afficher() {
    super.afficher();
    print('Service: $service');
  }
}

class PDG extends Manager {
  String societe;

  PDG(String nom, String prenom, DateTime dateNaissance, double salaire,
String service, this.societe)
      : super(nom, prenom, dateNaissance, salaire, service);

  @override
  void afficher() {
    super.afficher();
    print('Société: $societe');
  }
}

void main() {
  List<Entite> entites = [
    Travailleur('Travailleur 1', 'Prenom 1', DateTime(1990, 1, 1), 2000),
    Travailleur('Travailleur 2', 'Prenom 2', DateTime(1995, 2, 2), 2500),
    Travailleur('Travailleur 3', 'Prenom 3', DateTime(1992, 3, 3), 1800),
    Travailleur('Travailleur 4', 'Prenom 4', DateTime(1997, 4, 4), 2200),
    Travailleur('Travailleur 5', 'Prenom 5', DateTime(1993, 5, 5), 1900),
    Manager('Manager 1', 'Prenom 6', DateTime(1980, 6, 6), 3000, 'Service 1'),
    Manager('Manager 2', 'Prenom 7', DateTime(1985, 7, 7), 2800, 'Service 2'),
    PDG('PDG', 'Prenom 8', DateTime(1975, 8, 8), 5000, 'Service 3', 'Société
A'),
```

apcpedagogie.com

```
  ];

  print('Affichage avec for:');
  for (int i = 0; i < entites.length; i++) {
    entites[i].afficher();
    print('');
  }

  print('Affichage avec forEach:');
  entites.forEach((entite) {
    entite.afficher();
    print('');
  });
}
```

Cours Dart

Source : https://apcpedagogie.com/polymorphisme-dans-dart · téléchargé le 07/11/2026 · © APC Pédagogie · apcpedagogie.com

apcpedagogie.com