Programmation orientée objet : Encapsulation et héritage

https://github.com/heig-vd-progim-course/heig-vd-progim2-course

Visualiser le contenu complet sur GitHub à cette adresse.

V. Guidoux, avec l'aide de GitHub Copilot.

Plus de détails sur GitHub

Cette présentation est un résumé du contenu complet disponible sur GitHub.

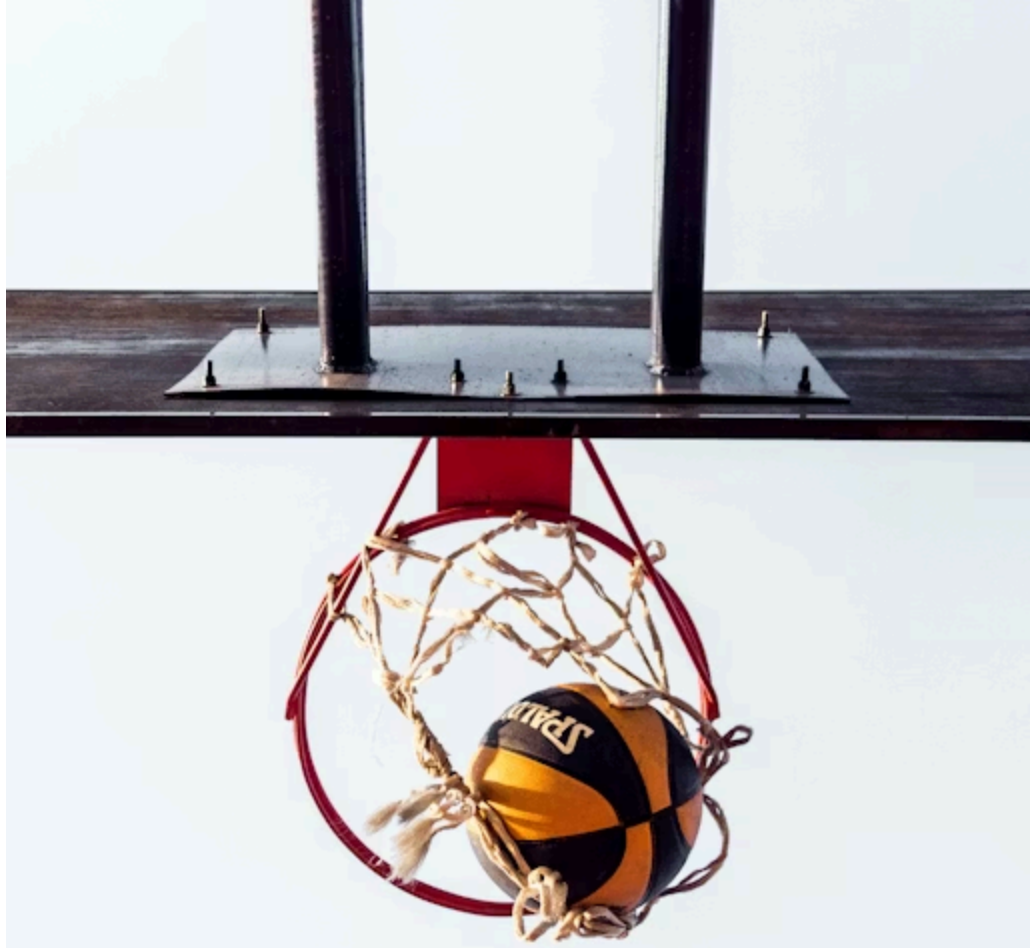
Pour plus de détails, consulter le contenu complet sur GitHub ou en cliquant sur l'en-tête de ce document.

Objectifs (1/5)

- Appliquer le principe d'encapsulation pour cacher l'implémentation interne.
- Valider les données dans les setters pour garantir la cohérence.
- Concevoir des classes avec une interface publique claire.
- Justifier les choix de visibilité des membres d'une classe.

Objectifs (2/5)

- Expliquer le concept d'héritage et sa finalité.
- Créer des classes dérivées en utilisant le mot-clé `extends`.
- Utiliser le mot-clé `super` pour appeler le constructeur de la classe parent.
- Identifier les relations "est-un" entre classes.

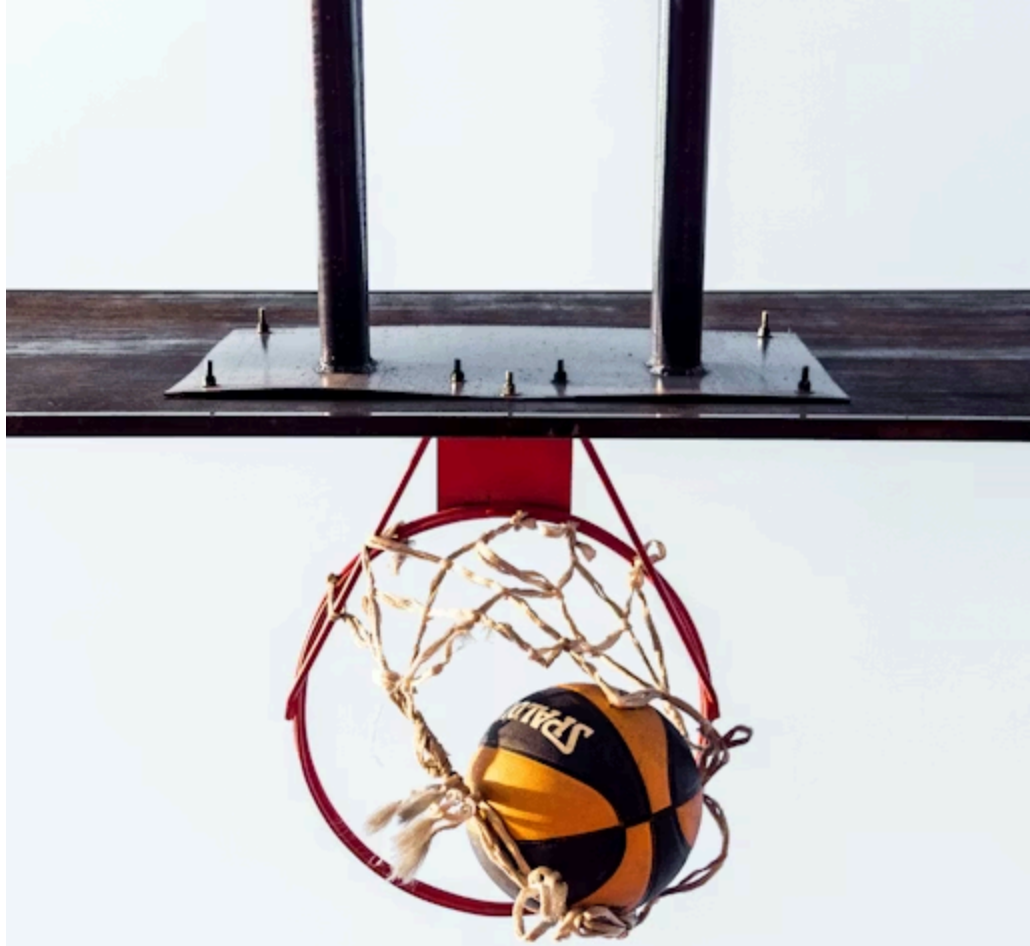


Objectifs (3/5)

- Organiser une hiérarchie de classes logique.
- Définir une classe abstraite avec le mot-clé `abstract`.
- Créer des méthodes abstraites à implémenter dans les sous-classes.
- Différencier une classe abstraite d'une classe concrète.

Objectifs (4/5)

- Justifier l'utilisation de classes abstraites pour factoriser du code.
- Appliquer le modificateur `protected` pour les membres accessibles aux sous-classes.
- Utiliser le mot-clé `final` pour empêcher la modification ou la redéfinition.



Objectifs (5/5)

- Évaluer quand utiliser `final` sur des classes, méthodes ou variables.

Rappel : Classes et objets

Dans la session précédente, nous avons créé des classes simples avec des attributs **directement accessibles** (sans encapsulation).

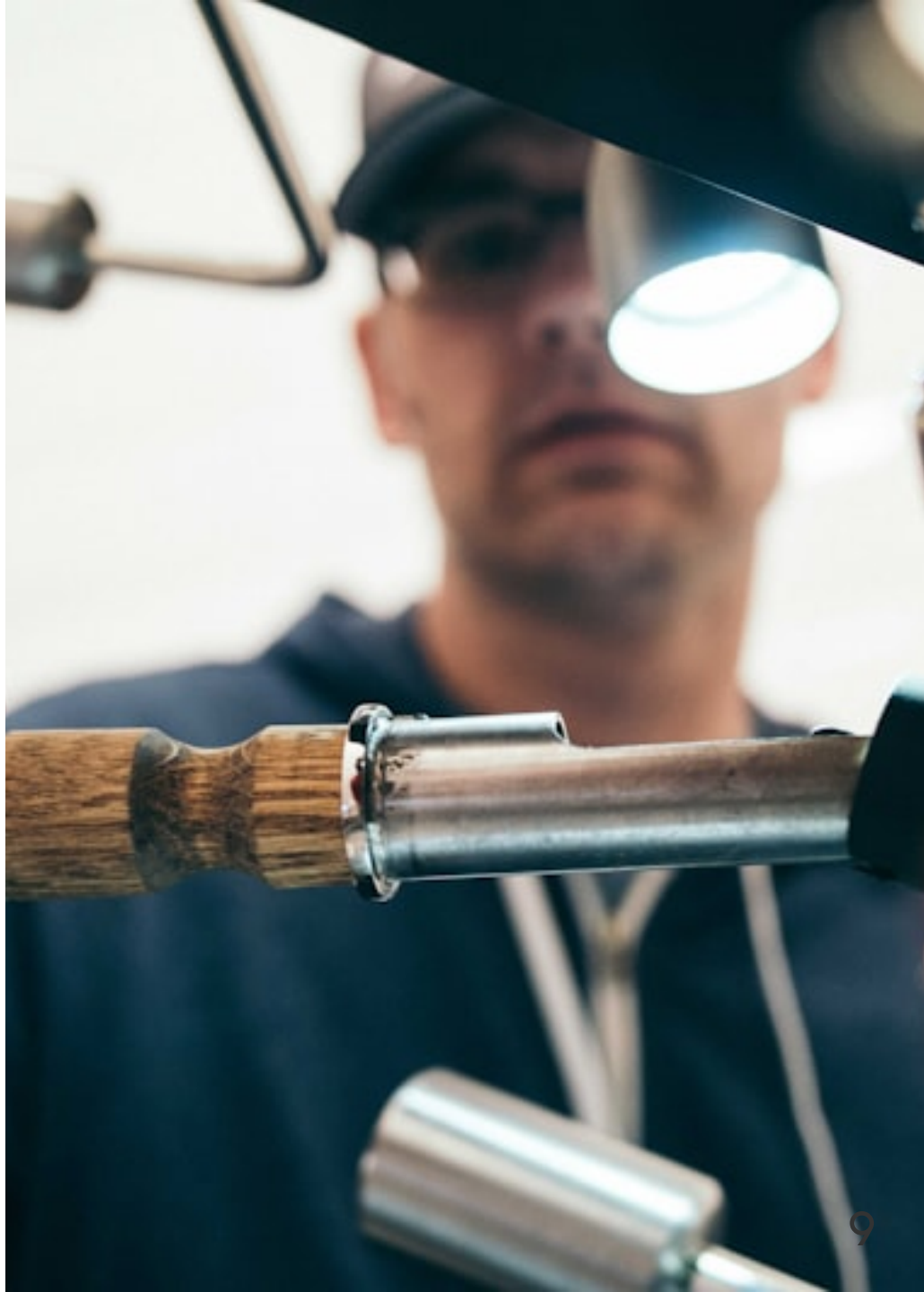
Problèmes :

- N'importe qui peut modifier directement les données.
- Difficile de contrôler les modifications.
- Code difficile à maintenir.
- Duplication de code entre classes similaires.

Solutions : Encapsulation et héritage

Encapsulation : protéger les données et contrôler leur accès.

Héritage : réutiliser du code et créer des hiérarchies de classes.



L'encapsulation

Protéger les données et contrôler leur accès

Problème sans encapsulation

```
class BankAccount {
    public String owner;
    public double balance;
}
```

```
BankAccount account = new BankAccount();
account.balance = -5000.0;  // Solde négatif !
account.balance = 999999999; // Montant irréaliste !
```

Aucune protection : modifications directes non contrôlées.

Bénéfices de l'encapsulation

- **Protection des données** : empêche les modifications directes non contrôlées.
- **Validation** : vérifie que les valeurs sont valides.
- **Maintenabilité** : facilite les modifications futures.
- **Interface claire** : définit ce qui peut être fait avec un objet.

Les modificateurs d'accès

Modificateur	Visibilité	Usage typique
`public`	Accessible de partout	Méthodes publiques, classes
`private`	Accessible uniquement dans la classe	Attributs, méthodes internes
`protected`	Accessible dans classe et sous-classes	Héritage
par défaut	Accessible dans le même package	Classes utilitaires internes

Rendre les attributs privés

```
class BankAccount {
    private String owner;
    private double balance;

    public void displayInfo() {
        System.out.println("Solde: " + balance + " CHF");
    }
}

// ... dans une autre classe

BankAccount account = new BankAccount();
account.balance = 1000.0; // ERREUR : balance est privé
```

Les getters et setters

```
class BankAccount {
    private String owner;
    private double balance;

    // Getters
    public String getOwner() { return owner; }
    public double getBalance() { return balance; }

    // Setters
    public void setOwner(String owner) { this.owner = owner; }
    public void setBalance(double balance) { this.balance = balance; }
}
```

Validation des données

```
public void setBalance(double balance) {
    if (balance < 0) {
        System.out.println("Erreur: solde négatif.");
        return;
    }
    this.balance = balance;
}

public void withdraw(double amount) {
    if (amount > balance) {
        System.out.println("Erreur: solde insuffisant.");
        return;
    }
    balance -= amount;
}
```

Le modificateur final

```
class Circle {
    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    // Pas de setRadius() car radius est final
}
```

Utile pour : constantes, identifiants uniques, valeurs qui ne doivent pas changer.

L'héritage

Réutiliser du code et créer des hiérarchies

Problème sans héritage

```
class Car {
    private String brand;
    private String model;
    private int year;
    private int numberOfDoors;
}

class Motorcycle {
    private String brand; // Code dupliqué
    private String model; // Code dupliqué
    private int year; // Code dupliqué
    private boolean hasSidecar;
}
```

Solution avec héritage

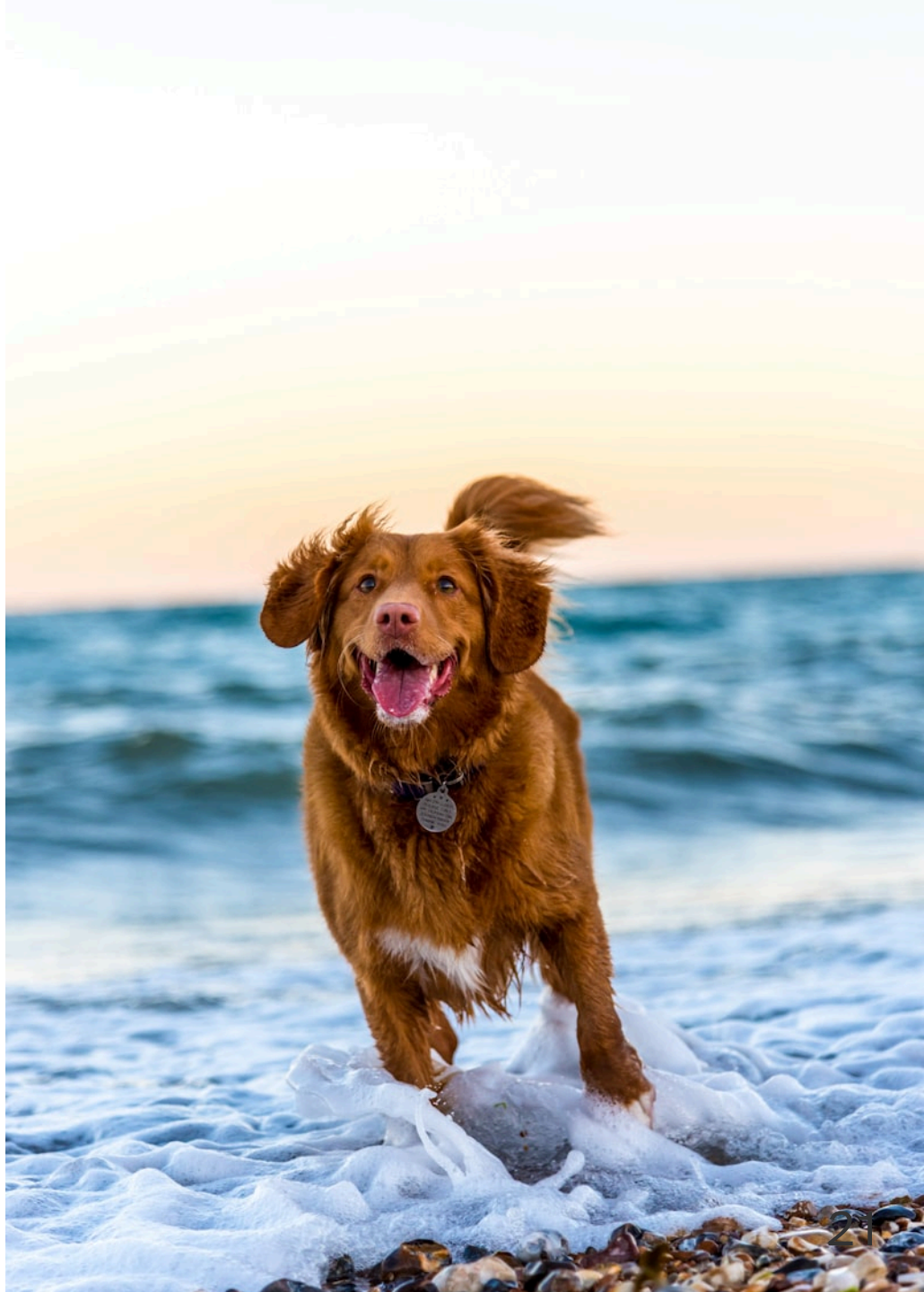
```
class Vehicle {
    protected String brand;
    protected String model;
    protected int year;
}

class Car extends Vehicle {  // Hérite de brand, model, year
    private int numberOfDoors;
}

class Motorcycle extends Vehicle { // Hérite de brand, model, year
    private boolean hasSidecar;
}
```

Bénéfices de l'héritage

- **Réutilisation du code** : évite la duplication.
- **Organisation logique** : hiérarchie qui reflète le monde réel.
- **Maintenabilité** : modifications propagées automatiquement.
- **Extensibilité** : facile d'ajouter de nouveaux types.



La relation "est-un"

L'héritage modélise une relation **"est-un"** (is-a) :

- Une voiture **est un** véhicule.
- Un chien **est un** animal.
- Une rose **est une** fleur.

Attention : ne pas confondre avec "a-un" (has-a) :

- Une voiture **a un** moteur (composition, pas héritage).

Créer une sous-classe avec extends

```
class Animal {
    protected String name;
    protected int age;

    public void eat() {
        System.out.println(name + " mange.");
    }
}

class Dog extends Animal {
    private String breed;

    public void bark() {
        System.out.println(name + " aboie !");
    }
}
```

Le mot-clé super

Appeler le constructeur parent :

```
class Dog extends Animal {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);  // Appelle Animal(name, age)
        this.breed = breed;
    }
}
```

`super()` doit être la **première instruction** du constructeur.

Le modificateur protected

```
class Parent {
    private String secret;      // Non accessible aux sous-classes
    protected String family;    // Accessible aux sous-classes
    public String forEveryone;  // Accessible partout
}

class Child extends Parent {
    public void test() {
        // secret = "test";  // ERREUR
        family = "test";     // OK
        forEveryone = "test"; // OK
    }
}
```

Les méthodes abstraites

Une **méthode abstraite** n'a pas d'implémentation :

```
abstract class Shape {
    protected String color;

    public abstract double calculateArea(); // Pas de corps

    public void displayColor() {
        System.out.println("Couleur: " + color);
    }
}
```

Les sous-classes **doivent** l'implémenter.

Implémenter les méthodes abstraites

```
class Circle extends Shape {
    private double radius;

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    private double width, height;

    @Override
    public double calculateArea() {
        return width * height;
    }
}
```

Les classes abstraites

Une **classe abstraite** ne peut pas être instanciée :

```
abstract class Employee {
    protected String name;
    public abstract double calculateSalary();
}

class FullTimeEmployee extends Employee {
    @Override
    public double calculateSalary() { return baseSalary; }
}

Employee e = new Employee(); // ERREUR !
FullTimeEmployee e = new FullTimeEmployee(); // OK
```

Classe abstraite vs concrète

Classe abstraite	Classe concrète
Mot-clé `abstract`	Pas de `abstract`
Ne peut pas être instanciée	Peut être instanciée
Peut avoir méthodes abstraites	Toutes méthodes implémentées
Sert de modèle	Utilisable directement

Redéfinition vs surcharge (1/2)

Redéfinition (overriding) :

- Même nom, **mêmes paramètres**, dans une **sous-classe**.
- Utilise `@Override`.

Surcharge (overloading) :

- Même nom, **paramètres différents**, dans la **même classe**.
- Pas d'annotation.

Redéfinition vs surcharge (2/2)

```
class Calculator {
    public int add(int a, int b) { return a + b; }         // Surcharge
    public double add(double a, double b) { return a + b; } // Surcharge
}
```

Le modificateur final (suite)

Sur une méthode : empêche la redéfinition dans les sous-classes.

```
class Parent {
    public final void criticalMethod() {
        // Ne peut pas être redéfinie
    }
}
```

Sur une classe : empêche l'héritage.

```
public final class MathUtils { /* Aucune sous-classe possible */ }
```

Organiser une hiérarchie de classes

Principes :

1. Identifier les caractéristiques communes.
2. Appliquer la relation "est-un".
3. Utiliser des classes abstraites pour les concepts généraux.
4. Créer des sous-classes pour les spécialisations.
5. Éviter les hiérarchies trop profondes (>3-4 niveaux).

Ressources

Documentation officielle :

- Java Inheritance (Oracle)
- Java Encapsulation (Oracle)

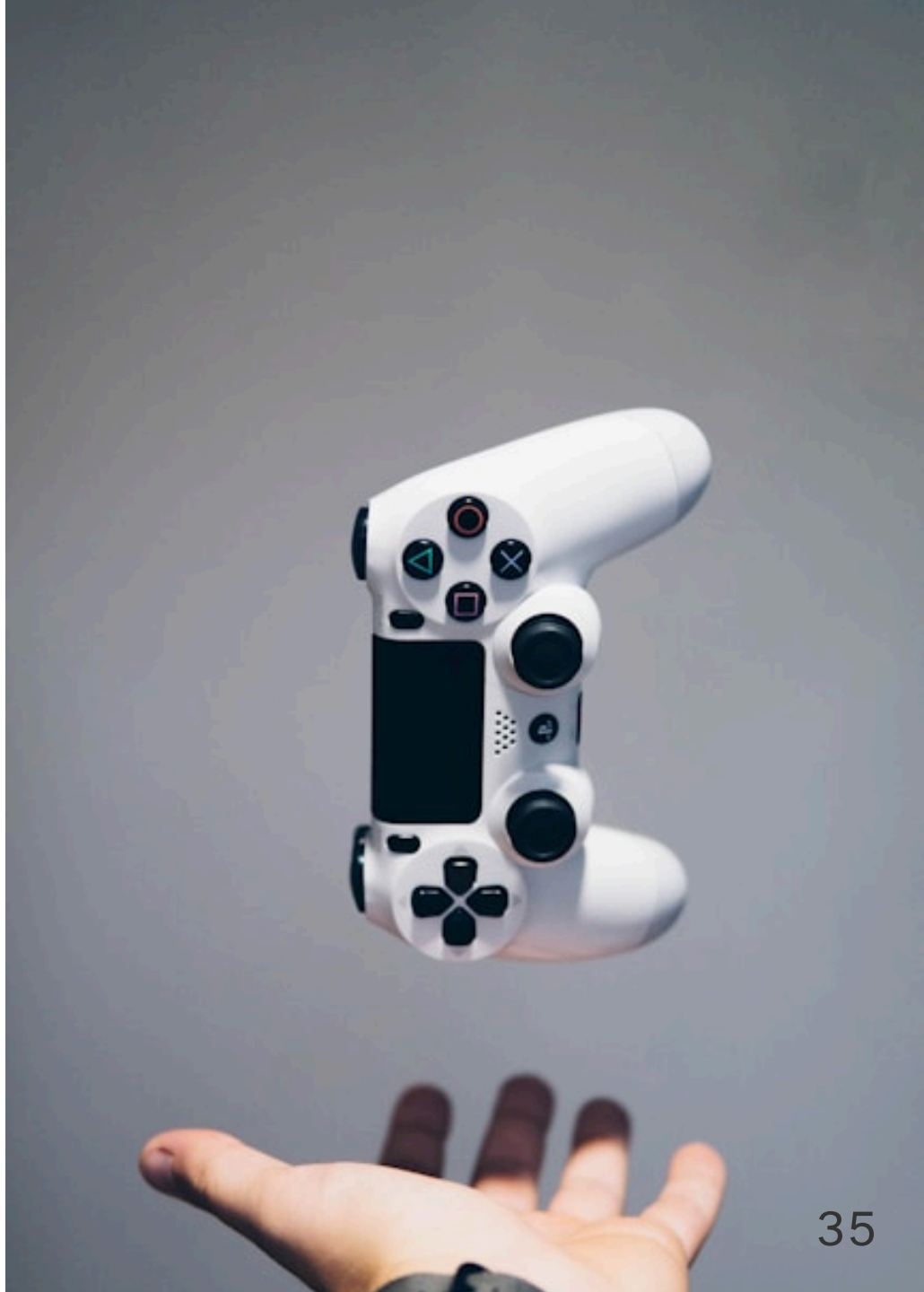
Tutoriels :

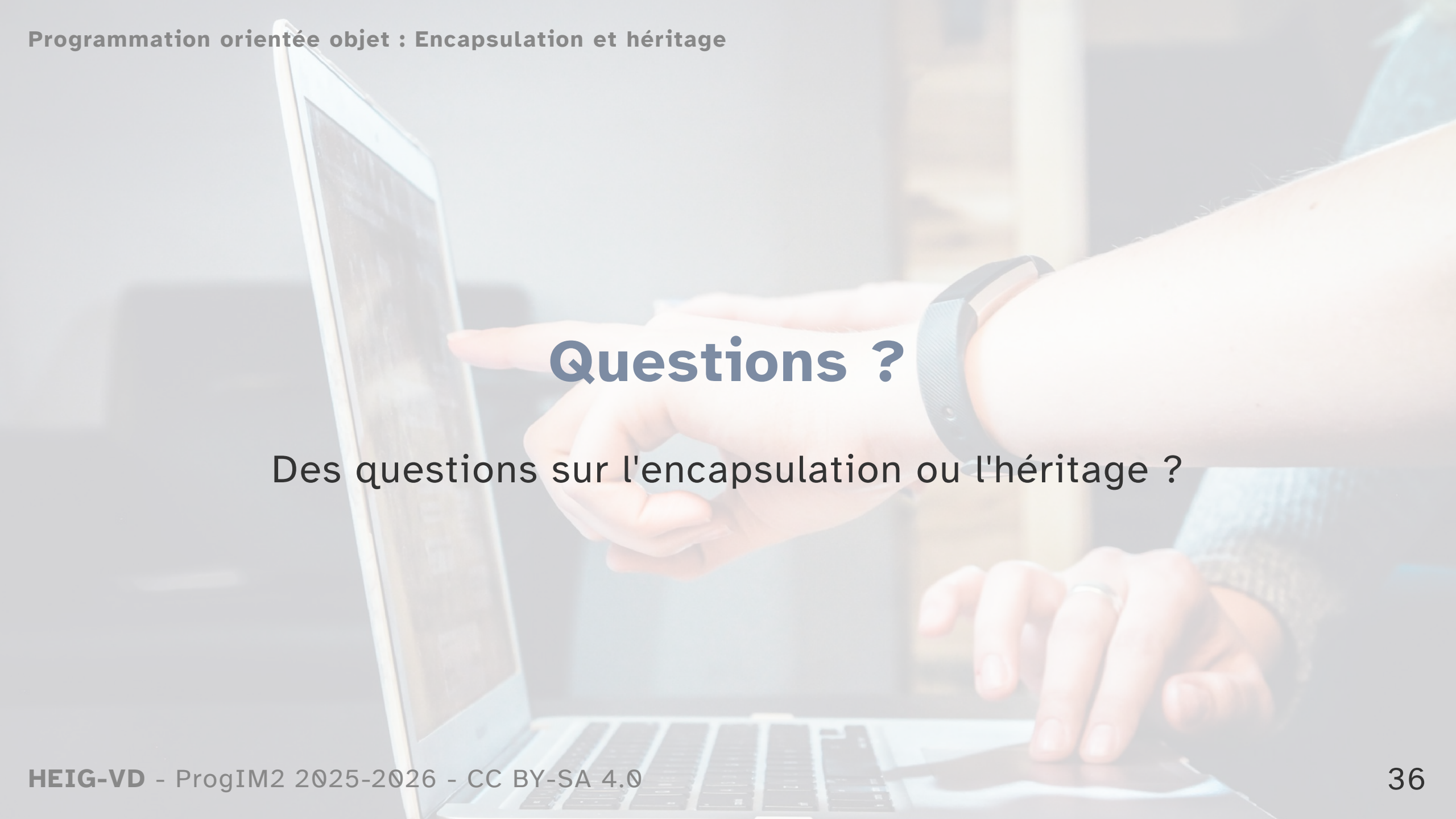
- W3Schools Java Tutorial
- Java Modifiers

À vous de jouer !

- (Re)lire le contenu de cours.
- Lire les exemples de code et les descriptions.
- Faire les exercices.
- Faire le mini-projet.
- Poser des questions si nécessaire.
- Entraidez-vous

➡️ Contenu complet sur GitHub.





Questions ?

Des questions sur l'encapsulation ou l'héritage ?

Sources

- Illustration principale par Luca Bravo sur Unsplash
- Illustration par Aline de Nadai sur Unsplash
- Illustration par Nikita Kachanovsky sur Unsplash