Programmation orientée objet : Polymorphisme

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 (1/4)

- Utiliser l'opérateur `instanceof` pour vérifier le type d'un objet.
- Effectuer un cast (conversion de type) de manière sécurisée.
- Identifier les limites de l'utilisation excessive de `instanceof`.
- Expliquer le concept de polymorphisme en POO.

Objectifs (2/4)

- Utiliser des références de type parent pour des objets de type enfant.
- Appliquer le polymorphisme pour traiter différents objets de manière uniforme.
- Démontrer comment le polymorphisme améliore la flexibilité du code.

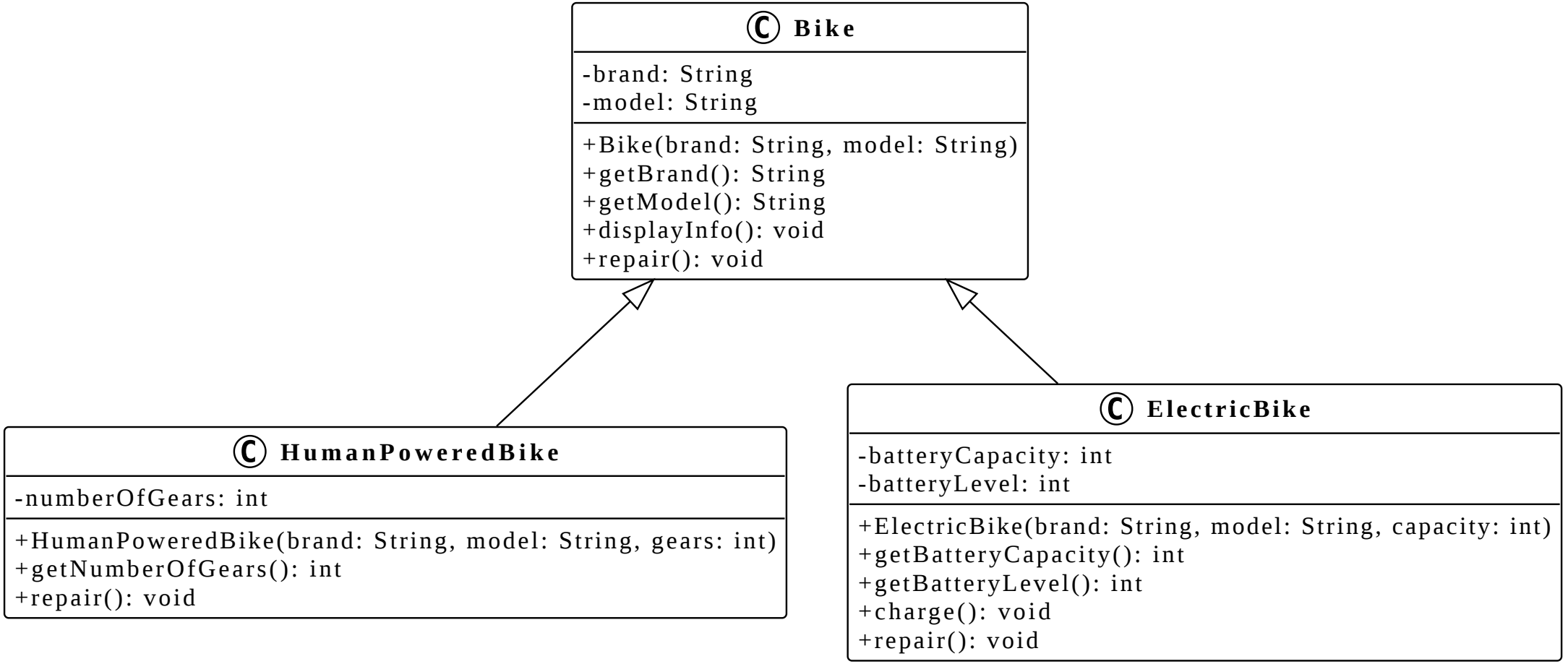
Objectifs (3/4)

- Appliquer la redéfinition pour adapter le comportement aux sous-classes.
- Définir une interface Java avec le mot-clé `interface`.
- Implémenter une ou plusieurs interfaces dans une classe.
- Différencier une interface d'une classe abstraite.

Objectifs (4/4)

- Justifier l'utilisation d'interfaces pour le polymorphisme.
- Redéfinir la méthode `toString()` pour représenter un objet sous forme de chaîne.
- Implémenter `equals()` pour comparer deux objets de manière significative.
- Implémenter `hashCode()` en cohérence avec `equals()`.

Hiérarchie des classes de vélos



L'opérateur instanceof

L'opérateur `instanceof` permet de vérifier si un objet est une instance d'un type particulier.

```
Bike bike = new ElectricBike("VanMoof", "S3", 500);

boolean isElectric = bike instanceof ElectricBike;      // true
boolean isHumanPowered = bike instanceof HumanPoweredBike; // false
boolean isBike = bike instanceof Bike;                  // true
```

Important : `instanceof` retourne `false` si l'objet est `null` (pas d'exception).

Utilisation avec le cast

Après vérification avec `instanceof`, on peut convertir la référence pour accéder aux méthodes spécifiques.

```
public void manageBike(Bike bike) {
    if (bike instanceof ElectricBike) {
        ElectricBike electric = (ElectricBike) bike;
        electric.charge();
    } else if (bike instanceof HumanPoweredBike) {
        HumanPoweredBike classic = (HumanPoweredBike) bike;
        System.out.println("Vitesses : " + classic.getNumberOfGears());
    }
}
```

Les limites de instanceof (1/2)

Bien que `instanceof` soit utile, son utilisation excessive révèle un problème de conception :

1. Code verbeux : cascade de `if-else` difficile à lire.

2. Violation du principe ouvert/fermé : ajouter un nouveau type oblige à modifier le code existant.

3. Duplication : même structure répétée partout.

Les limites de instanceof (2/2)

4. Couplage fort : le code doit connaître tous les types possibles.

5. Risque d'erreurs : oublier la vérification provoque une `ClassCastException`.

C'est précisément le problème que le polymorphisme résout de manière élégante.

Qu'est-ce que le polymorphisme ? (1/2)

Le terme vient du grec *poly* (plusieurs) et *morphe* (forme).

Définition : capacité d'un même code de manipuler des objets de types différents de manière uniforme.

Qu'est-ce que le polymorphisme ? (2/2)

Au lieu de :

```
if (bike instanceof HumanPoweredBike) {
    ((HumanPoweredBike) bike).repair();
} else if (bike instanceof ElectricBike) {
    ((ElectricBike) bike).repair();
}
```

On écrit simplement :

```
bike.repair();  // Appelle la bonne méthode automatiquement
```

Les trois piliers de la POO

Ces trois concepts travaillent ensemble :

1. Encapsulation : protège les données derrière des méthodes publiques.

2. Héritage : crée des classes à partir de classes existantes (relation "est un").

3. Polymorphisme : traite des objets différents de manière uniforme via une interface commune.

Références de type parent (1/2)

Une variable peut avoir un type déclaré différent du type réel de l'objet qu'elle référence.

```
Bike bike1 = new HumanPoweredBike("Decathlon", "Riverside", 21);
Bike bike2 = new ElectricBike("VanMoof", "S3", 500);
```

Tous les types héritent de `Bike`, donc ils **sont** des vélos.

Références de type parent (2/2)

Cette capacité permet de stocker différents types dans une même collection :

```
Bike[] fleet = new Bike[2];
fleet[0] = new HumanPoweredBike("Decathlon", "Riverside", 21);
fleet[1] = new ElectricBike("VanMoof", "S3", 500);

for (Bike bike : fleet) {
    bike.displayInfo();  // Méthode appropriée pour chaque type
}
```

Liaison dynamique

Quand on appelle une méthode sur une référence de type parent, Java utilise la **liaison dynamique**.

```
Bike bike = new ElectricBike("VanMoof", "S3", 500);
bike.displayInfo();  // Appelle la version de ElectricBike
```

La décision se fait **à l'exécution** (runtime), pas à la compilation.

Java regarde le **type réel** de l'objet (`ElectricBike`), pas le type de la référence (`Bike`).

Avantages du polymorphisme (1/2)

1. Code plus court et plus clair

```
// Au lieu de 4 méthodes différentes
public void repairBike(Bike bike) {
    bike.repair();  // Une seule méthode
}
```

2. Extensibilité

Ajouter un nouveau type (`TandemBike`) ne nécessite aucune modification du code existant.

Avantages du polymorphisme (2/2)

3. Réduction de la duplication

La logique est écrite une fois et fonctionne pour tous les types.

4. Abstraction du type concret

Le code manipule des concepts abstraits (un vélo) plutôt que des détails concrets (vélo électrique, cargo).

Redéfinition de méthodes

La redéfinition (*override*) permet à une sous-classe de fournir sa propre implémentation d'une méthode héritée.

```
abstract class Bike {
    public abstract void repair();
}

class ElectricBike extends Bike {
    @Override
    public void repair() {
        System.out.println("Vérification batterie et moteur.");
    }
}
```

Règles de la redéfinition (1/2)

Pour redéfinir correctement une méthode :

1. Même signature : même nom, même types et nombre de paramètres.

2. Type de retour compatible : même type ou sous-type (*covariant*).

3. Visibilité égale ou plus grande : `public` reste `public`, ne peut pas devenir `private`.

Règles de la redéfinition (2/2)

4. Exceptions plus spécifiques : peut lever les mêmes exceptions ou plus spécifiques, pas plus générales.

5. Méthodes non final : seules les méthodes non `final` peuvent être redéfinies.

Si une règle n'est pas respectée, le compilateur génère une erreur.

L'annotation @Override

L'annotation `@Override` marque explicitement qu'une méthode redéfinit une méthode héritée.

```
@Override
public void repair() {
    System.out.println("Réparation spécifique.");
}
```

- Vérification à la compilation (détecte les fautes de frappe).
- Documentation claire de l'intention.
- Protection contre les changements de signature.

Qu'est-ce qu'une interface ?

Une interface est un **contrat** qui spécifie des méthodes à implémenter, sans définir comment.

```
public interface Electric {
    int getBatteryLevel();
    void charge();
}
```

L'interface dit : "Si tu implémentes `Electric`, tu dois fournir ces méthodes".

Définir une interface

Caractéristiques d'une interface :

```
public interface Rideable {
    double getMaxSpeed();
}
```

- Méthodes abstraites par défaut (pas d'implémentation).
- Constantes possibles (`public static final`).
- Pas de constructeur.
- Pas de variables d'instance.

Implémenter une interface

Une classe implémente une interface avec le mot-clé `implements`.

```
class ElectricBike extends Bike implements Electric, Rideable {
    private int batteryLevel;

    @Override
    public int getBatteryLevel() { return batteryLevel; }

    @Override
    public void charge() { batteryLevel = 100; }

    @Override
    public double getMaxSpeed() { return 25.0; }
}
```

Implémenter plusieurs interfaces

Contrairement à l'héritage (une seule classe parent), une classe peut implémenter **plusieurs interfaces**.

```
class ElectricBike extends Bike implements Electric, Rideable {
    // Implémente toutes les méthodes de Electric et Rideable
}
```

C'est une des forces majeures des interfaces : composer des comportements de manière flexible.

Polymorphisme avec les interfaces

Une variable de type interface peut référencer tout objet qui implémente cette interface.

```
Electric bike = new ElectricBike("VanMoof", "S3", 500);

// Traitement uniforme
bike.charge();
```

Collections polymorphes avec interfaces

Les interfaces permettent de regrouper les objets par capacités.

```
Electric[] electricBikes = new Electric[2];
electricBikes[0] = new ElectricBike("VanMoof", "S3", 500);
electricBikes[1] = new ElectricBike("Stromer", "ST5", 983);

for (Electric bike : electricBikes) {
    bike.charge();
}
```

Traitement basé sur ce qu'ils **peuvent faire**, pas ce qu'ils **sont**.

Interface vs classe abstraite

Interface

- définit des capacités ("peut faire").
- Pour des classes non liées.
- Héritage multiple possible.
- Pas de code commun.

Classe abstraite

- définit une nature ("est un").
- Pour une hiérarchie de classes.
- Héritage simple uniquement.
- Partage de code commun.

La méthode toString()

```
@Override
public String toString() {
    return "Bike{brand='" + brand + "', model='" + model + "'}";
}
```

Appelée automatiquement par `System.out.println(object)`.

`toString()` retourne une représentation textuelle d'un objet.

Par défaut : `ElectricBike@15db9742` (peu utile).

Redéfinie : `Bike{brand='VanMoof', model='S3'}` (clair et utile).

Les méthodes equals() et hashCode()

Ces méthodes travaillent ensemble pour la comparaison d'objets.

```java
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null) return false;
    Bike that = (Bike) obj;
    return brand.equals(that.brand) && model.equals(that.model);
}

@Override
public int hashCode() {
    return brand.hashCode() + 31 * model.hashCode();
}
```

Pourquoi equals() et hashCode() ensemble ?

Règle : si vous redéfinissez `equals()`, vous **devez** redéfinir `hashCode()`.

Pourquoi : deux objets égaux doivent avoir le même `hashCode()`.

Sans cela, `HashSet` et `HashMap` ne fonctionnent pas correctement.

Collections polymorphes en pratique

Stocker et manipuler différents types dans une même collection :

```
Bike[] fleet = new Bike[2];
fleet[0] = new HumanPoweredBike("Decathlon", "Riverside", 21);
fleet[1] = new ElectricBike("VanMoof", "S3", 500);

for (Bike bike : fleet) {
    bike.displayInfo();  // Chaque vélo affiche ses infos
}
```

Conception flexible (1/2)

Sans polymorphisme (rigide) :

```
public void repairFleet(HumanPoweredBike[] classic,
                        ElectricBike[] electric) {
    for (HumanPoweredBike bike : classic) bike.repair();
    for (ElectricBike bike : electric) bike.repair();
}
```

Ajouter un type oblige à modifier cette méthode.

Conception flexible (2/2)

Avec polymorphisme (flexible) :

```
public void repairFleet(Bike[] bikes) {
    for (Bike bike : bikes) {
        bike.repair();
    }
}
```

Ajouter un nouveau type ne nécessite **aucune modification**.

Principe ouvert/fermé : ouvert à l'extension, fermé à la modification.

Questions

Est-ce que vous avez des questions ?

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.
- Entraider-vous !

➡️ Visualiser le contenu complet sur GitHub.

Sources (1/2)

- Illustration principale par Markus Winkler sur Unsplash
- Illustration par Aline de Nadai sur Unsplash

Sources (2/2)

- Illustration par Nikita Kachanovsky sur Unsplash