R201 — Développement orienté objets

Héritage

Clément Pagès
clement.pages@u-pec.fr

Université Paris-Est Créteil
BUT Informatique 1re année

2025-2026

Introduction

- On souhaite transmettre des attributs/méthodes entre les classes...

Introduction

- On souhaite transmettre des attributs/méthodes entre les classes...
- ... sans avoir à tout reprogrammer !

Introduction

- On souhaite transmettre des attributs/méthodes entre les classes...
- ... sans avoir à tout reprogrammer !

Avec la classe `Personnage`

- Les Gaulois et les Romains sont des personnages.

Introduction

- On souhaite transmettre des attributs/méthodes entre les classes...
- ... sans avoir à tout reprogrammer !

Avec la classe `Personnage`

- Les Gaulois et les Romains sont des personnages.
- Les classes `Gaulois` et `Romain` héritent de `Personnage`.

Introduction

- On souhaite transmettre des attributs/méthodes entre les classes...
- ... sans avoir à tout reprogrammer !

Avec la classe `Personnage`

- Les Gaulois et les Romains sont des personnages.
- Les classes `Gaulois` et `Romain` héritent de `Personnage`.
- Les attributs et méthodes de `Gaulois` et `Romain` auront automatiquement les attributs et méthodes de `Personnage`.

Introduction

- On souhaite transmettre des attributs/méthodes entre les classes...
- ... sans avoir à tout reprogrammer !

Avec la classe `Personnage`

- Les Gaulois et les Romains sont des personnages.
- Les classes `Gaulois` et `Romain` héritent de `Personnage`.
- Les attributs et méthodes de `Gaulois` et `Romain` auront automatiquement les attributs et méthodes de `Personnage`.
- On pourra ajouter des attributs et méthodes supplémentaires.

Sommaire

Le mot-clé `extends`

Personnage.java

```
public class Personnage {
    // …
}
```

Gaulois.java

```
public class Gaulois extends Personnage {
    private double tailleMoustache;
    // ...
}
```

IrreductibleGaulois.java

```
public class IrreductibleGaulois extends Gaulois {
    public void boirePotionMagique() {
        // ...
    }
    // ...
}
```

Romain.java

```
public class Romain extends Personnage {
    private String grade;
    // ...
}
```

Attributs hérités

Limites de la visibilité `private`

- La visibilité `private` restreint les attributs à la classe actuelle.
- Les classes filles ne peuvent pas les manipuler.

Attributs hérités

Limites de la visibilité `private`

- La visibilité `private` restreint les attributs à la classe actuelle.
- Les classes filles ne peuvent pas les manipuler.

Gaulois.java

```
public String toString() {
        return (this.nom + ", " + this.age + " ans, " +
                " moustache de " + this.tailleMoustache + "cm");
    }
```

Terminal

```
> javac Gaulois.java
Gaulois.java:19: error: nom has private access in Personnage
        return (this.nom + ", " + this.age + " ans, " + ...
                    ^
Gaulois.java:19: error: age has private access in Personnage
        return (this.nom + ", " + this.age + " ans, " +
                " moustache de " + this.tailleMoustache + "cm");
                                          ^
```

Attributs hérités

Limites de la visibilité `private`

- La visibilité `private` restreint les attributs à la classe actuelle.
- Les classes filles ne peuvent pas les manipuler.

Gaulois.java

```
public String toString() {
        return (this.nom + ", " + this.age + " ans, " +
                " moustache de " + this.tailleMoustache + "cm");
    }
```

Terminal

```
> javac Gaulois.java
Gaulois.java:19: error: nom has private access in Personnage
        return (this.nom + ", " + this.age + " ans, " + ...
                    ^
Gaulois.java:19: error: age has private access in Personnage
        return (this.nom + ", " + this.age + " ans, " +
                " moustache de " + this.tailleMoustache + "cm");
                                           ^
```

- La solution : visibilité `protected`.

Attributs hérités

Visibilité `protected`

Personnage.java

```
public class Personnage { // Commence par une majuscule
    /* Attributs : variables de classe */
    protected int age;    // private : attribut privé
    protected String nom; // private : attribut privé

    private static int nbPersonnages = 0;

    /* Constructeurs, getters, setters , méthodes, … */
}
```

Gaulois.java

```
public class Gaulois extends Personnage {
    private double tailleMoustache;
    /* Constructeurs (cf. plus bas, etc.) */

    public String toString() {
        return (this.nom + ", " + this.age + " ans, " +
                " moustache de " + this.tailleMoustache + "cm");
    }
}
```

Méthodes héritées

Principe général

- Les méthodes `public` et `protected` dans une classe mère sont applicables aux objets des classes filles.
- Évite d'avoir à programmer plusieurs fois la même chose.

Méthodes héritées

Principe général

- Les méthodes `public` et `protected` dans une classe mère sont applicables aux objets des classes filles.
- Évite d'avoir à programmer plusieurs fois la même chose.
- C'est le cœur de la programmation par objets.

Méthodes héritées

Principe général

- Les méthodes `public` et `protected` dans une classe mère sont applicables aux objets des classes filles.
- Évite d'avoir à programmer plusieurs fois la même chose.
- C'est le cœur de la programmation par objets.

Personnage.java

```
public String sePresenter() {
    return "Je m'appelle " + this.name + ".";
}
```

Main.java

```
public static void main(String[] args){
        Gaulois asterix   = new Gaulois("Asterix");

        System.out.println(asterix.sePresenter());
    }
}
```

Méthodes héritées

Principe général

- Les méthodes `public` et `protected` dans une classe mère sont applicables aux objets des classes filles.
- Évite d'avoir à programmer plusieurs fois la même chose.
- C'est le cœur de la programmation par objets.

Personnage.java

```
public String sePresenter() {
    return "Je m'appelle " + this.name + ".";
}
```

Main.java

```
public static void main(String[] args){
        Gaulois asterix   = new Gaulois("Asterix");

        System.out.println(asterix.sePresenter());
    }
}
```

- L'objet `asterix` est de classe `Gaulois`, qui hérite de `Personnage`.
- La méthode `sePresenter()` s'applique à l'objet `asterix`.

Méthodes héritées

Constructeurs

Le mot-clé `super`

- Idée : avoir des constructeurs pour les classes filles, qui ne nécessient pas de reprogrammer les constructeurs parents.
- Les objets hérités possèdent déjà les attributs parents.

Méthodes héritées

Constructeurs

Le mot-clé `super`

- Idée : avoir des constructeurs pour les classes filles, qui ne nécessient pas de reprogrammer les constructeurs parents.
- Les objets hérités possèdent déjà les attributs parents.

Gaulois.java

```
public Gaulois() {
    super(); // Constructeur de la classe mère sans paramètre
}

public Gaulois(String nom_) {
    super(nom_); // Constructeur de la classe mère,
                 // avec paramètre nom_
    this.tailleMoustache = 3.5; // Attribut spécifique Gaulois
}
```

Méthodes héritées

Redéfinition de méthodes

- Dans une classe fille, on souhaite modifier une méthode héritée (pour spécifier son fonctionnement).

Méthodes héritées

Redéfinition de méthodes

- Dans une classe fille, on souhaite modifier une méthode héritée (pour spécifier son fonctionnement).
- La méthode redéfinie a le même nom, les mêmes attributs que la méthode mère.

Méthodes héritées

Redéfinition de méthodes

- Dans une classe fille, on souhaite modifier une méthode héritée (pour spécifier son fonctionnement).
- La méthode redéfinie a le même nom, les mêmes attributs que la méthode mère.
- Mais son code est différent.

Méthodes héritées

Redéfinition de méthodes

- Dans une classe fille, on souhaite modifier une méthode héritée (pour spécifier son fonctionnement).
- La méthode redéfinie a le même nom, les mêmes attributs que la méthode mère.
- Mais son code est différent.
- Elle peut avoir un type de retour différent.

Méthodes héritées

Redéfinition de méthodes

- Dans une classe fille, on souhaite modifier une méthode héritée (pour spécifier son fonctionnement).
- La méthode redéfinie a le même nom, les mêmes attributs que la méthode mère.
- Mais son code est différent.
- Elle peut avoir un type de retour différent.
- Annotation `@Override` avant la redéfinition.

Méthodes héritées

Redéfinition de méthodes

- Dans une classe fille, on souhaite modifier une méthode héritée (pour spécifier son fonctionnement).
- La méthode redéfinie a le même nom, les mêmes attributs que la méthode mère.
- Mais son code est différent.
- Elle peut avoir un type de retour différent.
- Annotation `@Override` avant la redéfinition.
- Avantage : si un objet est créé pendant l'exécution avec une sous-classe qui n'est pas connue à l'avance, la bonne méthode sera appelée automatiquement.

Méthodes héritées

Redéfinition de méthodes

Personnage.java

```
public class Personnage {
    protected String nom;

    public Personnage(String nom_) { this.nom = nom_; }
    public String sePresenter() {
        return "Je m'appelle " + this.name + ".";
    }
}
```

Gaulois.java

```
public class Gaulois extends Personnage {

    public Personnage(String nom_) { super(nom_); }

    @Override
    public String sePresenter() {
        return "Je m'appelle " + this.name +
                ". Je suis Gaulois !";
    }
}
```

Sommaire

Classes abstraites

Principe général

- Permet de définir des classes qui *ne sont jamais instanciées*.
- Servent de racine pour des sous-classes.
- Il est possible d'implémenter certaines méthodes, mais pas toutes.

Classes abstraites

Principe général

- Permet de définir des classes qui *ne sont jamais instanciées*.
- Servent de racine pour des sous-classes.
- Il est possible d'implémenter certaines méthodes, mais pas toutes.

Exemple

La classe `Personnage` est abstraite. On ne crée des instances que de `Roamain` ou de `Gaulois`

Classes abstraites

Principe général

- Permet de définir des classes qui *ne sont jamais instanciées*.
- Servent de racine pour des sous-classes.
- Il est possible d'implémenter certaines méthodes, mais pas toutes.

Exemple

La classe `Personnage` est abstraite. On ne crée des instances que de `Roamain` ou de `Gaulois`

Personnage.java

```
public abstract class Personnage {
    protected String nom;
    public abstract sePresenter(String nom); // Méthode abstraite
    public string getNom() {
        return this.nom;
    }
}
```

Classes abstraites

Mutatis mutandis !

- Les classes et méthodes abstraites sont indiquées par le mot-clé `abstract`.
- Une classe doit être abstraite si elle contient au moins une méthode abstraite.
- On ne peut pas instancier des classes abstraites…

Classes abstraites

Mutatis mutandis !

- Les classes et méthodes abstraites sont indiquées par le mot-clé **`abstract`**.
- Une classe doit être abstraite si elle contient au moins une méthode abstraite.
- On ne peut pas instancier des classes abstraites...
- Mais on peut manipuler la classe générique pour les opérations compatibles.

Romain.java

```
public class Romain extends Personnage {
    public Romain(String nom_) {
        super(nom_);
    }

    public String sePresenter() {
        return "Je m'appelle " + this.nom +
                "et je ne suis pas fou !";
    }
}
```

Classes abstraites

Mutatis mutandis !

- Les classes et méthodes abstraites sont indiquées par le mot-clé **`abstract`**.
- Une classe doit être abstraite si elle contient au moins une méthode abstraite.
- On ne peut pas instancier des classes abstraites...
- Mais on peut manipuler la classe générique pour les opérations compatibles.

Gaulois.java

```
public class Gaulois extends Personnage {
    private double tailleMoustache;
    public Gaulois(String nom_, double tailleMoustache_) {
        super(nom_);
        this.tailleMoustache = tailleMoustache_;
    }
    public Gaulois(String nom_) {
        super(nom_);
        this.tailleMoustache = 3.5;
    }
    public String sePresenter() {
        return "Je m'appelle " + this.nom + ". Je suis Gaulois !";
    }
}
```

Classes abstraites

Mutatis mutandis !

- Les classes et méthodes abstraites sont indiquées par le mot-clé **`abstract`**.
- Une classe doit être abstraite si elle contient au moins une méthode abstraite.
- On ne peut pas instancier des classes abstraites...
- Mais on peut manipuler la classe générique pour les opérations compatibles.

Main.java

```
public class Main {
    public static void main(String[] args){
        Random generator = new Random() ;
        Personnage mystere ;
        if (generator.nextBoolean())
            mystere = new Gaulois("Asterix") ;
        else
            mystere = new Romain("Caligula Minus") ;
        System.out.println(mystere.sePresenter()) ;
    }
}
```

Les interfaces

Principe et motivation

- Certains personnages (de classe `Gaulois` ou `Romain`) sont des combattants, mais pas tous...

Héritage multiple

Cette approche s'appelle l'héritage multiple.

Les interfaces

Principe et motivation

- Certains personnages (de classe `Gaulois` ou `Romain`) sont des combattants, mais pas tous...
- On pourrait créer des sous-classe `SoldatRomain` et `IrreductibleGaulois`, qui héritent de `Romain` et de `Gaulois`...

Héritage multiple

Cette approche s'appelle l'héritage multiple.

Les interfaces

Principe et motivation

- Certains personnages (de classe `Gaulois` ou `Romain`) sont des combattants, mais pas tous...
- On pourrait créer des sous-classe `SoldatRomain` et `IrreductibleGaulois`, qui héritent de `Romain` et de `Gaulois`...
- Et qui héritent aussi d'une classe `Combattant`.

Héritage multiple

Cette approche s'appelle l'héritage multiple.

Les interfaces

Principe et motivation

- Certains personnages (de classe `Gaulois` ou `Romain`) sont des combattants, mais pas tous...
- On pourrait créer des sous-classe `SoldatRomain` et `IrreductibleGaulois`, qui héritent de `Romain` et de `Gaulois`...
- Et qui héritent aussi d'une classe `Combattant`.

Héritage multiple

Cette approche s'appelle l'héritage multiple.

Les interfaces

Principe et motivation

- Certains personnages (de classe `Gaulois` ou `Romain`) sont des combattants, mais pas tous...
- On pourrait créer des sous-classe `SoldatRomain` et `IrreductibleGaulois`, qui héritent de `Romain` et de `Gaulois`...
- Et qui héritent aussi d'une classe `Combattant`.

Héritage multiple

Cette approche s'appelle l'héritage multiple. Cela n'existe pas en Java.

Les interfaces

Principe et motivation

- Certains personnages (de classe `Gaulois` ou `Romain`) sont des combattants, mais pas tous...
- On pourrait créer des sous-classe `SoldatRomain` et `IrreductibleGaulois`, qui héritent de `Romain` et de `Gaulois`...
- Et qui héritent aussi d'une classe `Combattant`.

Héritage multiple

Cette approche s'appelle l'héritage multiple. Cela n'existe pas en Java.

- Solution : les interfaces.
- Contiennent uniquement des signatures de méthodes.

Les interfaces

Principe et motivation

- Certains personnages (de classe `Gaulois` ou `Romain`) sont des combattants, mais pas tous...
- On pourrait créer des sous-classe `SoldatRomain` et `IrreductibleGaulois`, qui héritent de `Romain` et de `Gaulois`...
- Et qui héritent aussi d'une classe `Combattant`.

Héritage multiple

Cette approche s'appelle l'héritage multiple. Cela n'existe pas en Java.

- Solution : les interfaces.
- Contiennent uniquement des signatures de méthodes.
- Sortes de « généralisation » de la notion classe abstraite.
- Une classe qui implémente une interface est obligée d'implémenter toutes les méthodes de l'interface.

Les interfaces

En pratique

Combattant.java

```
public interface Combattant {
    public void attaquer(Personnage p);
    public void seDefendre(Combattant c);
    public default void fuire() { // Implémentation par défaut
        System.out.println("Ne tapez plus !");
    }
}
```

Les interfaces

En pratique

Combattant.java

```
public interface Combattant {
    public void attaquer(Personnage p);
    public void seDefendre(Combattant c);
    public default void fuire() { // Implémentation par défaut
        System.out.println("Ne tapez plus !");
    }
}
```

Les attributs dans les classes abstraites

Les attributs d'une classe abstraite sont automatiquement `public static final`.

Les interfaces

En pratique

Combattant.java

```
public interface Combattant {
    public void attaquer(Personnage p);
    public void seDefendre(Combattant c);
    public default void fuire() { // Implémentation par défaut
        System.out.println("Ne tapez plus !");
    }
}
```

Les attributs dans les classes abstraites

Les attributs d'une classe abstraite sont automatiquement `public static final`.

- Les attributs d'interfaces sont constants.

Les interfaces

En pratique

Combattant.java

```
public interface Combattant {
    public void attaquer(Personnage p);
    public void seDefendre(Combattant c);
    public default void fuire() { // Implémentation par défaut
        System.out.println("Ne tapez plus !");
    }
}
```

Les attributs dans les classes abstraites

Les attributs d'une classe abstraite sont automatiquement `public static final`.

- Les attributs d'interfaces sont constants.
- Donc leur visibilité peut être `public`.

Les interfaces

En pratique

Combattant.java

```
public interface Combattant {
    public void attaquer(Personnage p);
    public void seDefendre(Combattant c);
    public default void fuire() { // Implémentation par défaut
        System.out.println("Ne tapez plus !");
    }
}
```

Les attributs dans les classes abstraites

Les attributs d'une classe abstraite sont automatiquement `public static final`.

- Les attributs d'interfaces sont constants.
- Donc leur visibilité peut être `public`.
- Ce sont nécessairement des attributs de classe.

Les interfaces

Implémenter une interface

IrreductibleGaulois

```
public class IrreductibleGaulois extends Gaulois
implements Combattant {
    private int force;
    private double potionMagiqueRestante;
    public IrreductibleGaulois(String nom_, double

    tailleMoustache_, int force_, double potionMagique_) {
        // ...
    }
    public void attaquer(Personnage p) {
        if(this.boirePotionMagique())
            System.out.println("Attaquer");
    }
    public void seDefendre(Combattant c) {
        this.esquive();
        this.attaquer(c); // On contre-attaque !
    }
}
```