ircam Centre Pompidou

UPMC SORBONNE UNIVERSITÉS

JAVA
- Polymorphisme -

Ninon Devis: ninon.devis@ircam.fr

Philippe Esling: esling@ircam.fr

License 3 Professionnelle - Multimédia

Retour sur l’héritage

```java
class Weapon {
    protected double damage;
    public Weapon ( double damage ) {
        this.damage = damage;
    }
}

class Axe extends Weapon {
    private static final double DAMAGE = 10;
    public Axe() {
        super (DAMAGE);
    }

}
    class Hammer extends Weapon {
    private static final double DAMAGE = 20;
    public Hammer () {
        super(DAMAGE);
    }
}
```

- Quelles sont les variables d’instance ?
- Quelles sont les variables de classe?

Rappels & Quiz

- *Quelles sont les variables d'instance*: pas déclarées avec static, elles sont allouées à l'instance, accessible avec "point" si elles sont public: Weapon.damage
- *Quelles sont les variables de classe*: déclarées avec static, sans instance, accessibles avec "point": Axe.DAMAGE, Hammer.DAMAGE

➔ Variable d'instance différente pour chaque nouvel objet.

➔ Variable de classe identique pour chaque objet, elle est attachée à la définition de la classe.

- *Qu'est ce qu'un constructeur* ? Ce n'est pas une méthode de l'objet mais une fonction qui construit l'objet.

Rappels & Quiz

- Qu'est ce que super? Qu'est ce que this ?

➔ Lors de la *construction*:

- super permet d'appeler un autre constructeur de la classe parente

  super(arg1, arg2)

- this permet d'appeler le constructeur de la classe courante

  this(arg1, arg2)

➔ Lors d'*opérations*: pour accéder à la référence de la classe parente ou courante en vue d'accéder à des attributs ou méthodes explicitement.

```java
class Weapon {
    protected double damage;
    public Weapon (double damage) {
        this.damage = damage;
    }
}
```

```java
class Weapon {
    protected double damage;
    public Weapon (double d) {
        damage = d;
    }
}
```

```java
class Weapon {
    protected double damage;
    public Weapon (double d) {
        this.damage = d;
    }
}
```

Rappels & Quiz

- *Pourquoi un appel de constructeur dans un constructeur doit-il être la première instruction ?*

➔ Afin de pouvoir allouer la mémoire nécessaire aux variables de la classe mère.

Coding Style en Java: *construire d'abord les classes de bases avant celles qui les étendent.*

- *Que se passe t'il si super n'est pas appelé dans une classe fille?*

➔ Java insère automatiquement `super()` au début => erreur de compilation si il n'y a pas de constructeur par défaut (i.e. sans argument) pour la classe mère.

Plan du cours

I. Type statiques et type dynamique

II. Polymorphisme
   A. Polymorphisme par sous-typage (overriding)
   B. Polymorphisme de surcharge (overloading)
   C. Mariage polymorphisme

III. La classe Object

Types statiques et dynamiques

Compile-time *Run-time*

```
Weapon hammer = new Hammer();
```

type statique

type dynamique

- C’est le type associé à la variable lors de la compilation.
- C’est le type de la déclaration de la variable.
- Les variables déclarées avec types primitifs ne peuvent être que statiques
  ➔ pas d’héritage

- C’est le type réel de la variable, tel qu’on l’a initialisé, déduit à l'exécution.
- Le type dynamique est toujours une sous-classe ou identique au type statique.

```
1 Axe axe = new Weapon(39);
```

➔ n’a pas de sens puisqu’une arme n’est pas une hache: peut être d’autres choses.

Type statique

Quels sont les types statiques de chaque variables?

```
1  class WeaponStore {
2      Weapon cheater = new Weapon(100);
3      Weapon axe = new Axe();
4      Weapon hammer = new Hammer();
5      int number_weapons = 3;
6      Number extra_damage = new Integer(42);
7
8      public int price(Weapon w) {/*...*/}
9  }
10
11 //...In main function.
12
13 WeaponStore store = new WeaponStore();
14 store.price(new Axe());
15 store.price(new Weapon(22));
```

→ Weapon
→ Weapon
→ Weapon
→ int
→ Number

→ Weapon for w

→ WeaponStore
→ the temporary variable has type Axe
→ temporary has type weapon

Type dynamique

Quels sont les types dynamiques de chaque variables?

```
1  class WeaponStore {
2      Weapon cheater = new Weapon(100);
3      Weapon axe = new Axe();
4      Weapon hammer = new Hammer();
5      int number_weapons = 3;
6      Number extra_damage = new Integer(42);
7
8      public int price(Weapon w) {/*...*/}
9  }
10
11 //...In main function.
12
13 WeaponStore store = new WeaponStore();
14 store.price(new Axe());
15 store.price(new Weapon(22));
```

→ Weapon

→ Axe

→ Hammer

→ int

→ Integer

→ w can be Weapon, Axe or Hammer

→ WeaponStore

→ the temporary variable has type Axe

→ temporary has type weapon

Mini-Projet

Role Player Game

- Exercice final = Text-based RPG
  - Un personnage peut acheter des armes.
  - Celles-ci permettront de se battre contre des monstres.

Partie 1:

- Le personnage pourra acheter une arme parmi celles proposées dans un magasin, elles infligeront des dommages différents en fonction de leur prix.

➔ Commencez par implémenter plusieurs classes d'armes.

➔ Implémentez ensuite une classe magasin d'armes.

- 20-30 minutes

Polymorphisme

- Concept fondamental et général en informatique.
- Signifie “*avoir plusieurs formes*”.
- Sujet récurrent en entretien d’embauche.
- Il en existe plusieurs:
  - Polymorphisme de coercition (casting).
  - Polymorphisme par sous-typage (via héritage et redéfinition).
  - Polymorphisme de surcharge. (overload)
  - Polymorphisme paramétrique (via les génériques). *(Cours 5)*

Polymorphisme

Polymorphisme de coercition: casting

```
1 double price = 9.99;
2 int rounded_price = (int) price;
3 // rounded_price = ?
```

➔ La conversion permet de considérer que price est de type `int` au lieu de `double`!

Polymorphisme

Polymorphisme par sous-typage

➔ Ajouter une méthode `ascii_art` à vos armes qui retourne un `String` contenant le dessin ASCII de l'arme. *(Vous pouvez trouver facilement les ASCII arts sur internet)*

- 10 minutes

```
1  class Bow { //...
2      public String ascii_art(){
3          return
4          "    (         \n" +
5          "     \        \n" +
6          "      )       \n" +
7          "##--------> \n" +
8          "      )       \n" +
9          "     /        \n" +
10         "    (         \n";
11     }
12 }
```

Polymorphisme

Polymorphisme par sous-typage

Soit un magasin d'arme `ArrayList<Weapon> store;`

pouvez-vous afficher le dessin ASCII de toutes les armes de ce magasin ?

➔ Pas de méthode `ascii_art` pour la classe `Weapon`…

➔ Comment faire pour utiliser cette classe sous sa forme réelle, c'est à dire utiliser le type dynamique Axe, Hammer, Bow…?

```
1 class Weapon {
2     public String ascii_art(){
3         return ???;
4     }
5 }
```

Une arme n'a pas de dessin général c'est un concept abstrait !

➔ Il faut mettre à jour la classe Weapon en prenant en compte les besoins!

Polymorphisme

Polymorphisme par sous-typage

➔ La classe Weapon doit être une classe abstraite ! Pourquoi pas une interface?

- ◆ Elle contient des membres mais certaines méthodes n'ont pas de corps.

```java
abstract class Weapon {
    protected double damage ;
    public Weapon(double damage) {
        this.damage = damage;
    }
    abstract public String ascii_art();
}

abstract class Bow extends Weapon {
    private static final double DAMAGE = 40;
    public Bow {
        super(DAMAGE)
    }
    public String ascii_art(){
        return
        "   (        \n" +
        "    \       \n" +
        "     )      \n" +
        "##--------> \n" +
        "     )      \n" +
        "    /       \n" +
        "   (        \n";
    }
}

```

```java
class Hammer extends Weapon {
    private static final double DAMAGE = 30;
    public Hammer() {
        super(DAMAGE);
    }
    public String ascii_art () {
        return  "     _ _ \n" +
                " |_|_|\n" +
                "   |   \n" +
                "   |   \n";
    }
}

public class TestWeapon {
    public static void main (String[] args) {
        ArrayList <Weapon> store = new ArrayList<>();
        store.add(new Hammer());
        store.add(new Axe());
        for (Weapon w : store) {
            System.out.println (w.ascii_art());
        }
    }
}
```

Polymorphisme

Polymorphisme de surcharge (overloading)

- Créer une classe Monster et Obstacle possédant un certains nombre de points de vie et une méthode pour diminuer ces points de vie.
- Ajouter deux méthodes à vos classes d'armes pour attaquer les monstres et obstacles.
- Les dégâts infligés par vos armes sur les ennemis seront différents en fonction de l'arme et de l'ennemi (obstacle ou monstre).

➔ Attention au nom de la méthode pour diminuer les points de vie! Eviter set_* et get_*

- 15 minutes

Polymorphisme

Polymorphisme de surcharge (overloading)

```
1  class Monster {
2    private double life = 100;
3    public void hit_me(double damage) { life -= damage; }
4  }
5  class Obstacle { /* similar */ }
6
7  class Axe {
8    static final double MONSTER_DAMAGE_RATIO = 0.8;
9    static final double OBSTACLE_DAMAGE_RATIO = 1.2;
10
11   public void attack_monster(Monster m) {
12     m.hit_me(damage * MONSTER_DAMAGE_RATIO);
13   }
14
15   public void attack_obstacle(Obstacle o) {
16     o.hit_me(damage * OBSTACLE_DAMAGE_RATIO);
17   }
18 }
19 class Hammer { /* similar */ }
```

Quel est le problème avec cette solution ?

Ne jamais se répéter dans le code!

➔ Utiliser une classe parente Destructible qui factorise Monster et Obstacle

Polymorphisme

Polymorphisme de surcharge (overloading)

Exemple de classe parente Destructible:

```
1 class Destructible {
2   protected double life = 100;
3   public void hit_me(double damage) { life -= damage; }
4 }
5 class Monster extends Destructible { /* ... */}
6 class Obstacle extends Destructible { /* ... */ }
```

➔ Créer la classe Destructible

- 5 minutes

Polymorphisme

Polymorphisme de surcharge (overloading)

```java
public class Sum {
    public int sum(int x, int y)
    {
        return (x + y);
    }

    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    public double sum(double x, double y)
    {
        return (x + y);
    }

    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));       // 30
        System.out.println(s.sum(10, 20, 30));   // 60
        System.out.println(s.sum(10.5, 20.5));   // 31
    }
}
```

- La surcharge est un mécanisme statique permettant d'utiliser un même nom pour plusieurs signatures de méthodes lorsque celles-ci ont un rôle similaire.

- Se base uniquement sur le type statique, peu importe le type dynamique, les appels de méthodes sont résolus à la compilation (static binding ou polymorphisme statique).

Polymorphisme

Polymorphisme de surcharge (overloading)

Quelle est la méthode appelée, ou l'erreur si, pour chaque objet déclaré ci-dessous, on fait `axe.attack(object)` ?

L'overloading ne se base que sur le type statique!

```
1  class Axe {
2    public void attack(Monster m) {}        // (1)
3    public void attack(Obstacle o) {}       // (2)
4    public void attack(Destructible d) {}   // (3)
5  }
6
7
8  Destructible dmonster = new Monster();    // (3)
9  Destructible dobstacle = new Obstacle();  // (3)
10 Monster monster = new Monster();          // (1)
11 Obstacle obstacle = new Obstacle();       // (2)
```

Statique

Dynamique

Polymorphisme

Polymorphisme de surcharge (overloading)

Quelle est la méthode appelée ou l'erreur de compilation pour ces exemples ?

```
1  class Axe {
2    public void attack(Monster m, Obstacle o) {} // (1)
3    public void attack(Destructible d, Monster m) {} // (2)
4    public void attack(Monster m, Destructible d) {} // (3)
5  }
6
7  Destructible dmonster = new Monster();
8  Destructible dobstacle = new Obstacle();
9  Monster monster = new Monster();
10 Obstacle obstacle = new Obstacle();
11
12 axe.attack(monster, obstacle);    ➔ (1)
13 axe.attack(dobstacle, monster);   ➔ (2)
14 axe.attack(dobstacle, dmonster);  ➔ error: no such method
15 axe.attack(monster, dobstacle);   ➔ (3)
16 axe.attack(dmonster, dmonster);   ➔ error: no such method
17 axe.attack(monster, monster);     ➔ error: ambiguous call between (2) & (3)!
```

Polymorphisme

Polymorphisme de surcharge (overloading)

Comment savoir quelle méthode sera sélectionnée à la compilation ?

1. *Identifier* les classes à explorer (type statique de l'objet + hiérarchie).
2. *Localiser* les méthodes accessibles avec le même nom.
3. *Sélectionner* les méthodes avec la même nombre d'arguments.
4. *Sélectionner* les méthodes applicables: celles dont les types de arg1, …, argn sont "inférieurs ou égaux" aux types des paramètres.
5. *Déterminer* celle qui est la plus spécifique. *(cf slide suivante)*

Note : le type de retour n'entre pas en compte.

Polymorphisme

Polymorphisme de surcharge (overloading)

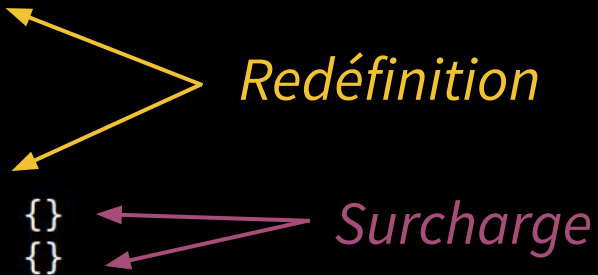
Comment déterminer la méthode qui sera la plus spécifique ? (étape 5)

1. Calculer la distance d'héritage entre les types d'arguments et les types de paramètres.
2. Additionner les distances.
3. La méthode avec la plus petite distance est sélectionnée.
4. Si plusieurs distances identiques alors erreur *Ambiguous call*.

Polymorphisme

Bilan

```
1 class Bow {
2   public void attack(Monster m, Obstacle o) {}
3 }
4
5 class Axe {
6   public void attack(Monster m, Obstacle o) {}
7   public void attack(Destructible d, Monster m) {}
8   public void attack(Monster m, Destructible d) {}
9 }
```



Polymorphisme

A retenir

Redéfinition (overriding)

- Un type de base peut avoir plusieurs formes (les sous-types).
- Le mécanisme d'*overriding* permet de redéfinir plus précisément un comportement d'une méthode qui est déjà fournie par la super-classe.
- La redéfinition se produit donc dans deux classes ayant une relation d'héritage.
- En cas de redéfinition de méthode les paramètres doivent être identiques.
- La méthode appelée est choisie à l'exécution (*late-binding*).

Surcharge (Overloading)

- Une méthode peut avoir plusieurs formes.
- Permet d'utiliser un même nom pour différentes implantations. Cependant la logique entre les différentes méthodes doit être similaire.
- La surcharge de méthode est effectuée dans la classe elle-même.
- En cas de surcharge de méthode, les paramètres doivent être différents.
- La méthode appelée est choisie à la compilation (*static-binding*).

Polymorphisme

Mariage Polymorphique

- On peut utiliser le polymorphique de surcharge et par sous-typage conjointement.
- On choisit d'abord la méthode avec le mécanisme d'*overloading* (sélectionnée à la compilation).
- Et puis on regarde si le mécanisme d’*overriding* s'applique (la signature doit être *override-equivalent* à celle choisie à la compilation).

Polymorphisme

Mariage Polymorphique - Exercice Avancé

```java
class A {
  void m(A x, B y){System.out.println ("1");}
  void m(B x, A y){System.out.println ("2");}
}
class B extends A {
  void m(B x, B y){System.out.println ("3");}
}
class C extends B {
  void m(B x, B y){System.out.println ("4");}
  void m(C x, C y){System.out.println ("5");}
  void m(B x, A y){System.out.println ("6");}
}
```

- Le type dynamique de b1 est B: on cherche les méthodes appelables dans la classe B et la classe plus générale A. Les réponses possibles sont donc (3), (2) et (1).
- Les types statiques des arguments sont respectivement C et B. On calcule la distance entre les types des arguments et des paramètres:
  (C, B) => (A, B) = 2
  (C, B) => (B, A) = 2
  (C, B) => (B, B) = 1
- La méthode sélectionnée est donc la (3)

```java
class MariagePolymorphique {
  public static void main(String[] args) {
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;

    a1.m(b1,c1);
    b1.m(b1,c1);
    c1.m(b1,c1);
    a1.m(a1,a1);

    a2.m(b1,c1);
    a3.m(b1,c1);
    b2.m(b1,c1);

    a1.m(b2,a3);
    a2.m(b2,a3);
    a3.m(b2,a3);

    a1.m(c1,b1);
    b1.m(c1,b1);
    b2.m(c1,b1);
    c1.m(c1,b1);
  }
}
```

Polymorphisme

Mariage Polymorphique - Exercice Avancé

Corrigé

```java
class A {
  void m(A x, B y){System.out.println ("1");}
  void m(B x, A y){System.out.println ("2");}
}
class B extends A {
  void m(B x, B y){System.out.println ("3");}
}
class C extends B {
  void m(B x, B y){System.out.println ("4");}
  void m(C x, C y){System.out.println ("5");}
  void m(B x, A y){System.out.println ("6");}
}
```

```java
class MariagePolymorphique {
  public static void main(String[] args) {
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;

    a1.m(b1,c1); // ambiguous between (1) and (2)
    b1.m(b1,c1); // (3)
    c1.m(b1,c1); // (4)
    a1.m(a1,a1); // no suitable method found

    a2.m(b1,c1); // ambiguous between (1) and (2)
    a3.m(b1,c1); // ambiguous between (1) and (2)
    b2.m(b1,c1); // (4)

    a1.m(b2,a3); // (2)
    a2.m(b2,a3); // (2)
    a3.m(b2,a3); // (6)

    a1.m(c1,b1); // ambiguous between (1) and (2)
    b1.m(c1,b1); // (3)
    b2.m(c1,b1); // (4)
    c1.m(c1,b1); // (4)
  }
}
```

Mini-Projet

Text-based RPG (cf. https://www.materiamagica.com/play/web/)

- Jeu RPG basé sur une interaction textuelle
  - Des menus de choix s'affichent sur la console
  - L'utilisateur choisit une action en tapant au clavier (Java: classe Scanner)
  - On doit créer son Personnage (nom, propriétés, argent, XP, mana)
    - Celui-ci peut-être choisi parmis un ensemble de castes (Sorcier, Elfe, ...)
    - Chaque type possède des caractéristiques différentes
  - Le système de jeu permet à tout moment
    - D'acheter des armes dans un magasin
    - Changer son inventaire (équipement, à minima les armes)
    - Se déplacer sur une carte virtuelle (matrice 2D): avant, arrière, gauche, droite
      - Le joueur commence en bas à gauche et doit arriver en haut à droite
      - Cette carte contient aléatoirement des monstres et obstacles
      - Lors de la rencontre avec des objets, on peut choisir de combattre ou fuir
      - Système de combat aléatoire donnant des points d'XP

La classe Object

- Qu’est ce que la classe `Object` ?
- Java est un langage “purement” objet, toutes les fonctions sont attachées à un objet (qu'elles soient statiques ou non).
- Toutes classes héritent de `Object`! (Sauf `Object` lui-même)

```
1 class Object {
2   protected Object clone() throws CloneNotSupportedException { ... }
3   public boolean equals(Object obj) { ... }
4   public String toString() { ... }
```

La classe Object

Exercice

- ➔ Ajoutez à `Weapon` les méthodes `clone`, `equals` et `toString` de `Object`.
- ➔ Créer deux objets `Weapon`, clonez-les et comparez-les.
- ➔ Peut-on éviter les *casts* ?
- ➔ Que se passe t-il si on fait `weapon.equals(new ArrayList())` ou `weapon.equals(null)` ?
- ➔ Pourquoi ne pas utiliser `public boolean equals(weapon w)` ?

- ◆ 20 minutes

La classe Object

Correction

➔ Ajoutez à `Weapon` les méthodes `clone`, `equals` et `toString` de `Object`.

```
1  abstract class Weapon {
2    protected double damage;
3    // ...
4
5    public Object clone() throws CloneNotSupportedException {
6      return new Weapon(damage);
7    }
8    public boolean equals(Object obj) {
9      return ((Weapon)obj).damage == damage;
10   }
11   public String toString() {
12     return "Weapon with damage: " + damage;
13   }
14 }
```

La classe Object

Correction

➔ Créer deux objets `Weapon`, clonez-les et comparez-les.

```
1 Weapon axe = new Weapon(10);
2 Weapon hammer = new Weapon(20);
3 Weapon axe_c = (Weapon)axe.clone();
4 Weapon hammer_c = (Weapon)hammer.clone();
5 if(axe_c.equals(hammer_c)) {
6   System.out.println(axe_c + " and " + hammer_c + " are equals.");
7 } else {
8   System.out.println(axe_c + " and " + hammer_c + " are differents.");
9 }
```

La classe Object

Correction

- Peut-on éviter les *casts* ?
  - On peut éviter ceux relatifs à la conversion du type de retour de clone.
  - Lors de l'overloading, le type de retour n'entre pas en compte. Lors de l'overriding, celui-ci peut être co-variant.
  - Les clauses throws n'entrent pas en compte dans overloading et peuvent être plus restrictives lors de l'overriding, donc on peut supprimer throws CloneNotSupportedException.
  -
- Que se passe t-il si on fait `weapon.equals(new ArrayList())` ou `weapon.equals(null)` ?
- Pourquoi ne pas utiliser `public boolean equals(weapon w)` ?

Type dynamique

Run-time

- C’est le type réel de la variable, tel qu’on l’a initialisé, déduit à l'exécution.

```
Weapon hammer = new Hammer();
```

type statique — type dynamique

- Le type dynamique est toujours une sous-classe ou identique au type statique.

```
1 Axe axe = new Weapon(39);
```

➔ n’a pas de sens puisqu’une arme n’est pas une hache: peut être d’autres choses.

Type statique

Compile-time

- C’est le type associé à la variable lors de la compilation.
- C’est le type de la déclaration de la variable.
- Les variables déclarées avec types primitifs ne peuvent être que statiques

➔ pas d’héritage