Cours 3 – Polymorphisme

Programmation objets, web et mobiles en Java
Licence 3 Professionnelle - Multimédia

Philippe Esling, Pierre Talbot (esling@ircam.fr)

Doctorant UPMC/IRCAM

14 octobre 2014

Le menu

Introduction

Indispensable

- Le polymorphisme est fondamentale dans le paradigme orienté-objet (mais pas que !).
- Sujet récurrent en entretien d'embauche !
- Beaucoup d'autres notions sont basées sur ces principes.

Avant-propos

Un exemple d'héritage dont on se servira par la suite.

```
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);
  }
}
```

Quizz

- Quelles sont les variables d'instance et de classe ?
- Qu'est-ce qu'un constructeur ?
- Qu'est-ce que `super` et `this` ?
- Pourquoi un appel de constructeur (dans un constructeur) doit-il être la première instruction ?
- Que se passe t'il si on n'appelle pas `super` ?

Solutions

Instance vs Classe

- D'instance : `Weapon.damage`. De classe : `Axe.DAMAGE` et `Hammer.DAMAGE`.
- Variable d'instance : différente pour chaque nouvel objet, "attachée" à une *instance* en particulier.
- Variable de classe : identique pour chaque objet, "attachée" à la définition de la classe (du moule), c'est une variable globale accessible uniquement via la classe englobante.

Solutions

Constructeur

- Un constructeur n'est pas une méthode de l'objet. C'est une fonction qui construit l'objet !
- C'est pour ça qu'ils ne sont pas hérités.
- En C par exemple, on a souvent des fonctions `make_something` qui retournent une structure.
- C'est similaire à des constructeurs.

Solutions

`super` et `this` : 2 usages

- Lors de la construction : appeler un autre constructeur, de la classe parente (`super(arg1, arg2)`) ou de 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 (`this.damage`).

Solutions

`super` et `this` : 2 usages

- Lors de la construction : appeler un autre constructeur, de la classe parente (`super(arg1, arg2)`) ou de 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 (`this.damage`).

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

Hiding de l'attribut `damage` par la variable locale.

Solutions

Constructeur en premier !

- Philosophie objet en Java (et en C++ aussi), on construit d'abord les classes de base avant celles qui les étendent.
- Parfois embêtant quand la classe a besoin d'une initialisation complexe du membre.

Solutions

Constructeur en premier !

- Philosophie objet en Java (et en C++ aussi), on construit d'abord les classes de base avant celles qui les étendent.
- Parfois embêtant quand la classe a besoin d'une initialisation complexe du membre.

Coding style

Utiliser une méthode statique si initialisation complexe. Ou une autre classe si initialisation en plusieurs temps (*builder pattern*), on y reviendra dans les cours 6/7 sur les *design patterns*.

Solutions

`super` n'est pas appelé

- Insertion automatique de `super()` au début.
- Erreur de compilation s'il n'y a pas de constructeur par défaut (sans argument) pour la classe de base.

Le menu

Type statique (compile-time)

Type statique

- 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).

Exemple : types statiques

```
class WeaponStore{
  Weapon cheater = new Weapon(100);
  Weapon axe = new Axe();
  Weapon hammer = new Hammer();
  int number_weapons = 3;
  Number extra_damage = new Integer(42);

  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore();
store.price(new Axe());
store.price(new Weapon(22));
```

Solution exemple : types statiques

```
class WeaponStore{
  Weapon cheater = new Weapon(100); // Weapon
  Weapon axe = new Axe(); // Weapon
  Weapon hammer = new Hammer(); // Weapon
  int number_weapons = 3; // int
  Number extra_damage = new Integer(42); // Number

  // The static type of w is Weapon
  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore(); // WeaponStore
store.price(new Axe()); // the temporary variable has type Axe.
store.price(new Weapon(22)); // temporary has type Weapon.
```

Type dynamique (run-time)

Type dynamique

- Le type réel de la variable, tel qu'on l'a initialisé, déduit à l'exécution.
- Le type dynamique (c_1) est toujours une sous-classe ou identique ($c_1 \leq c_2$) au type statique (c_2).
- Par exemple, `Axe axe = new Weapon(39);` n'a pas de sens. Une arme *n'est pas* une hache, ça peut être d'autres choses !
- Et puis, techniquement, comment serait initialisé les membres restants de `Axe` ?

Exemple : types dynamiques

```
class WeaponStore{
  Weapon cheater = new Weapon(100);
  Weapon axe = new Axe();
  Weapon hammer = new Hammer();
  int number_weapons = 3;
  Number extra_damage = new Integer(42);

  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore();
store.price(new Axe());
store.price(new Weapon(22));
```

Solution exemple : types dynamiques

```
class WeaponStore{
  Weapon cheater = new Weapon(100); // Weapon
  Weapon axe = new Axe(); // Axe
  Weapon hammer = new Hammer(); // Hammer
  int number_weapons = 3; // int
  Number extra_damage = new Integer(42); // Integer

  // The dynamic type of w can be
  // Weapon, Axe or Hammer.
  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore(); // WeaponStore
store.price(new Axe()); // the temporary variable has type Axe.
store.price(new Weapon(22)); // temporary has type Weapon.
```

Le menu

Polymorphisme

- Concept fondamental et général en informatique.
- Signifie "avoir plusieurs formes".
- Un même type peut se comporter de plusieurs façons.

Polymorphisme

- Concept fondamental et général en informatique.
- Signifie "avoir plusieurs formes".
- Un même type peut se comporter de plusieurs façons.

Plusieurs polymorphismes

- Polymorphisme de coercition (*casting*).
- Polymorphisme par sous-typage (via héritage et redéfinition).
- Polymorphisme de surcharge.
- Polymorphisme paramétrique (via les génériques).

Polymorphisme de coercition

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

Polymorphisme de coercition

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

Conversion

Une conversion permet de considérer que `price` est de type `int` au lieu de `double`.

Polymorphisme de coercition

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

Conversion

Une conversion permet de considérer que `price` est de type `int` au lieu de `double`.

- Notion valide dans beaucoup d'autres langages orientés objet (C++, Python, ...) ou non (C, Javascript, ...).
- Attention aux spécificités de chaque langage ! En Java, prendre en compte l'auto-boxing. En C++, 4 différents opérateurs de cast (`static_cast`, `dynamic_cast`,...).

Le menu

Polymorphisme par sous-typage

Défi introductif

Ajouter une méthode `ascii_art` qui retourne un `String` contenant le dessin ASCII de l'arme.

Polymorphisme par sous-typage

Défi introductif

Ajouter une méthode `ascii_art` qui retourne un `String` contenant le dessin ASCII de l'arme.

```
class Axe { // ...
  // from http://www.chris.com/ascii/index.php?art=objects/axes
  public String ascii_art() {
    return
      " /'-./\_    \n" + // What's wrong here?
      ":     ||,>  \n" +
      " \.-'||     \n" + // And here?
      "      ||    \n" +
      "      ||    \n" +
      "      ||    \n";
  }
}
```

Raw string et *multi-lines string* pas encore présents en Java.

Polymorphisme par sous-typage

Magasin

Soit un magasin d'arme `ArrayList<Weapon> store;`, pouvez-vous afficher le dessin ASCII de toutes les armes de ce magasin ?

Polymorphisme par sous-typage

Magasin

Soit un magasin d'arme `ArrayList<Weapon> store;`, pouvez-vous afficher le dessin ASCII de toutes les armes de ce magasin ?

Problèmes

- La classe `Weapon` n'a pas de méthode `ascii_art` !
- Comment faire pour utiliser la classe `Weapon` sous sa *forme* réel ? (type dynamique : `Axe` ou `Hammer`).

Mécanisme de redéfinition (*overriding*)

Signature *override-equivalent*

Deux signatures de méthodes sont *override-equivalent* si elles ont exactement le même nom et les mêmes types d'arguments et de retour. Le type de retour peut être co-variant (voir la partie sur `Object`).

Mécanisme de redéfinition (*overriding*)

Signature *override-equivalent*

Deux signatures de méthodes sont *override-equivalent* si elles ont exactement le même nom et les mêmes types d'arguments et de retour. Le type de retour peut être co-variant (voir la partie sur `Object`).

Overriding

Pour toutes classes $T \leq \mathit{Weapon}$, si une méthode $T.m$ est *override-equivalent* à $\mathit{Weapon}.m$, alors la méthode appelée sera celle de la plus petite sous-classe dont m est *override-equivalent*.

Mécanisme de redéfinition (*overriding*)

Signature *override-equivalent*

Deux signatures de méthodes sont *override-equivalent* si elles ont exactement le même nom et les mêmes types d'arguments et de retour. Le type de retour peut être co-variant (voir la partie sur `Object`).

Overriding

Pour toutes classes $T \leq \mathit{Weapon}$, si une méthode $T.m$ est *override-equivalent* à $\mathit{Weapon}.m$, alors la méthode appelée sera celle de la plus petite sous-classe dont m est *override-equivalent*.

Dynamique

Les appels de méthodes sont résolus à l'exécution (*late-binding*), en effet on ne peut pas deviner à la compilation le type dynamique de l'objet. Pourquoi ? Car il peut varier à l'exécution suivant les *inputs* d'un utilisateur.

Exemple *overriding*

```
class Weapon {
  public String ascii_art() {
    return ????;
  }
}
```

Exemple *overriding*

```
class Weapon {
  public String ascii_art() {
    return ????;
  }
}
```

Problème de conception ! Une arme n'a pas de dessin général. D'ailleurs est-ce qu'une arme "générale" peut exister ? Probablement pas, c'est un concept abstrait.

Exemple *overriding*

```
class Weapon {
  public String ascii_art() {
    return ????;
  }
}
```

Problème de conception ! Une arme n'a pas de dessin général. D'ailleurs est-ce qu'une arme "générale" peut exister ? Probablement pas, c'est un concept abstrait.

Refactoring

- On doit mettre à jour la classe `Weapon` en prenant en compte les nouveaux *besoins*.
- La classe `Weapon` doit être une classe abstraite ! Elle contient des membres (l'interface est donc exclue) mais certaines méthodes n'ont pas de corps.

Exemple complet

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

class Axe extends Weapon {
  private static final double DAMAGE = 10;
  public Axe() {
    super(DAMAGE);
  }
  public String ascii_art() {
    return
      "<|>\n" +
      " | \n" +
      " | \n";
  }
}
```

Exemple complet (suite)

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

Que retenir du polymorphisme par sous-typage ?

- Polymorphisme car un type de base peut avoir plusieurs formes (les sous-types).
- Mécanisme d'*overriding* permettant de redéfinir plus précisément un comportement.
- La méthode appelée est choisie à l'exécution (*late-binding*).
- À la compilation, la méthode est choisie via un autre mécanisme, un autre type de polymorphisme, l'*overloading*.

Le menu

Polymorphisme de surcharge (*overloading*)

Défi introductif

- Créer une classe `Monster` et `Obstacle` possédant un certain nombre de points de vie et une méthode pour diminuer ces points de vie.
- Ajouter ensuite deux méthodes à `Axe` et `Hammer` pour attaquer les monstres et obstacles.
- Les dégâts qu'une hache inflige aux monstres sont pondérés par 0.8, et aux obstacles par 1.2.
- Pour le marteau on a 1.4 et 0.7.

Réflexion sur nom de méthode

Avez-vous appelé la méthode pour diminuer les points de vie `set_life` ou similaire ?

Réflexion sur nom de méthode

Avez-vous appelé la méthode pour diminuer les points de vie `set_life` ou similaire ?

Coding style

- Les accesseurs `set_*` et `get_*` sont de très mauvais noms qui entraînent des pratiques violant tous les principes orienté-objets.
- Ils violent l'encapsulation et entraînent une exposition des membres internes.
- Une méthode doit rendre un *service*, ça doit transparaître dans le nom.
- C'est difficile de trouver des bons noms mais très important.
- Parfois on veut donner l'accès aux membres car la classe ne rend aucun service mais sert juste de container (voir PODs et POJO, `http://en.wikipedia.org/wiki/Plain_old_data_structure`).

Première solution

```
class Monster {
  private double life = 100;
  public void hit_me(double damage) { life -= damage; }
}
class Obstacle { /* similar */ }

class Axe {
  static final double MONSTER_DAMAGE_RATIO = 0.8;
  static final double OBSTACLE_DAMAGE_RATIO = 1.2;

  public void attack_monster(Monster m) {
    m.hit_me(damage * MONSTER_DAMAGE_RATIO);
  }

  public void attack_obstacle(Obstacle o) {
    o.hit_me(damage * OBSTACLE_DAMAGE_RATIO);
  }
}
class Hammer { /* similar */ }
```

Observation

```
public void attack_monster(Monster m)
```

Rien ne vous gêne ?

Observation

```
public void attack_monster(Monster m)
```

Rien ne vous gêne ?

Coding style

Il ne faut jamais se répéter, dans les noms, dans le code. La signature nous indique déjà qu'on attaque un monstre, pas la peine de le redire dans le nom.

Deuxième solution

```
class Monster {
  private double life = 100;
  public void hit_me(double damage) { life -= damage; }
}
class Obstacle { /* similar */ }

class Axe {
  static final double MONSTER_DAMAGE_RATIO = 0.8;
  static final double OBSTACLE_DAMAGE_RATIO = 1.2;

  public void attack(Monster m) {
    m.hit_me(damage * MONSTER_DAMAGE_RATIO);
  }

  public void attack(Obstacle o) {
    o.hit_me(damage * OBSTACLE_DAMAGE_RATIO);
  }
}
class Hammer { /* similar (constants change) */ }
```

Surcharge (Overloading)

Définition

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

Surcharge (Overloading)

Définition

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

Statique

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).

Surcharge (Overloading)

Lors d'un appel `obj.method(`a_1`, ..., `a_n`)`, comment savoir quelle méthode sera sélectionnée à la compilation ? (En gris, étapes triviales. . .).

1. Identifier les classes à explorer (type statique de `obj` + hiérarchie).
2. Localiser les méthodes *accessibles* avec le même nom.
3. Sélectionner les méthodes avec la même arité (nombre d'arguments).
4. Sélectionner les méthodes *applicables*, càd, dont les types de a_n sont $\leq T_n$, T_n étant les types des paramètres.
5. Appliquer un algorithme pour sélectionner *la plus spécifique*.

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

Algorithme de résolution pour l'*overloading*

L'algorithme peut être différent suivant le langage. Même entre différentes versions d'un même langage (Java 1.2 vs Java 1.5 ou sup.). On présente ici le plus récent pour Java.

1. Soit A_i les types des arguments et P_i les types des paramètres.
2. Pour chaque arguments calculer la distance "d'héritage" entre A_i et P_i, si $A_i \equiv P_i$ alors la distance est 1.
3. Additionner les distances.
4. La méthode avec la plus petite distance est sélectionnée.
5. Si plusieurs distances identiques alors erreur *Ambiguous call*.

Notes sur l'*overloading*

- Il est le plus souvent utilisé lorsque les méthodes sont non ambiguës, càd :
  - Une arité différente.
  - Les types de leurs paramètres sont indépendants (non liés par héritage).
- Sinon le programmeur doit lui-même exécuter l'algorithme de résolution dans sa tête pour savoir quelle méthode est appelée.
- C'est à utiliser avec précaution et à garder extrêmement simple.
- Généralement la philosophie adoptée dans les librairies Java.

Exercice I

Factorisation

Utiliser une classe parente `Destructible` qui factorise `Monster` et `Obstacle`.

Exercice I

Factorisation

Utiliser une classe parente `Destructible` qui factorise `Monster` et `Obstacle`.

Solution

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

Exercice II

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

```
class Axe {
  public void attack(Monster m) {} // (1)
  public void attack(Obstacle o) {} // (2)
  public void attack(Destructible d) {} // (3)
}

Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();
```

Solution : Exercice II

```
Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();

axe.attack(dmonster); // (3)
axe.attack(dobstacle); // (3)
axe.attack(monster); // (1)
axe.attack(obstacle); // (2)
```

Statique

Ne pas oublier que l'*overloading* ne se base que sur le type statique !

Exercice III

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

```
class Axe {
  public void attack(Monster m, Obstacle o) {} // (1)
  public void attack(Destructible d, Monster m) {} // (2)
  public void attack(Monster m, Destructible d) {} // (3)
}

Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();

axe.attack(monster, obstacle);
axe.attack(dobstacle, monster);
axe.attack(dobstacle, dmonster);
axe.attack(dmonster, dmonster);
axe.attack(monster, monster);
axe.attack(monster, dobstacle);
```

Solution : Exercice III

```
class Axe {
  public void attack(Monster m, Obstacle o) {} // (1)
  public void attack(Destructible d, Monster m) {} // (2)
  public void attack(Monster m, Destructible d) {} // (3)
}

Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();

axe.attack(monster, obstacle); // (1)
axe.attack(dobstacle, monster); // (2)
axe.attack(dobstacle, dmonster); // error : no such method
axe.attack(dmonster, dmonster); // error : no such method
axe.attack(monster, monster); // error : ambiguous call between
                              //         (2) and (3)
axe.attack(monster, dobstacle); // (3)
```

Que retenir du polymorphisme de surcharge ?

- Polymorphisme car une méthode peut avoir plusieurs formes (les autres méthodes du même nom).
- Mécanisme d'*overloading* permettant d'utiliser un même nom pour différentes implémentations. Néanmoins la logique entre les différentes méthodes doit être similaires !
- La méthode appelée est choisie à la compilation (*static-binding*).
- On peut marier l'*overloading* et l'*overriding*.

Le menu

Marier *overriding* et *overloading*

- 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).

Exercice

```
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");}
}
```

Exercice (suite)

Pour chaque appel, quelle est la méthode sélectionnée à la compilation, et puis à l'exécution ?

```
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);
    // ... (more in the next slide)
```

Exercice (suite)

```
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = 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);
  }
}
```

Correction exercice

```
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;

    // solution of the form '(compile-time) / (execution-time)'
    a1.m(b1,c1); // ambiguous between (1) and (2)
    b1.m(b1,c1); // (3)/(3)
    c1.m(b1,c1); // (4)/(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); // (3)/(4)

    a1.m(b2,a3); // (2)/(2)
    a2.m(b2,a3); // (2)/(2)
    a3.m(b2,a3); // (2)/(6)
    // ... (more in the next slide).
```

Correction exercice (suite)

```
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;

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

Exercices et ressources supplémentaires

Cours UPMC

- Notamment dans le cours “Typage et Polymorphisme” de M2 STL d’Emmanuel Chailloux, cours 4 notamment.
- `http://www-apr.lip6.fr/~chaillou/Public/enseignement/2013-2014/tep/`.
- Maintenant renommé “Typage et analyse statique” (plus large), `http://www-apr.lip6.fr/~chaillou/Public/enseignement/2014-2015/tas/`.
- Voir certains exercices d’examen.

Exercices et ressources supplémentaires

Référence langage officiel Java

- Lien : `http://docs.oracle.com/javase/specs/`, ğ suivant pour la version Java SE 8.
- ğ8.4.8 : *overriding*.
- ğ8.4.9 : *overloading*.
- ğ15.12 : Invocation d'une méthode (étapes effectuées par le compilateur).
- Difficile à lire et comprendre car tous les aspects du langage sont pris en considération.
- C'est néanmoins la meilleure solution pour transformer des doutes en certitudes.

Le menu

Tout est objet

- Java est un langage “purement” objet, toutes les fonctions sont attachées à un objet (qu'elles soient statiques ou non (méthodes)).
- Toutes classes héritent de `Object`.
- On appelle `Object` un *god object*.
- Le risque de ces objets est de cumuler trop de responsabilités et d'être trop chargés, ce qui viole le principe de responsabilité unique (SRP).

La classe Object

```
class Object {
  // Utility methods
  protected Object clone() throws CloneNotSupportedException { ... }
  public boolean equals(Object obj) { ... }
  public String toString() { ... }
  public int hashCode() { ... }
  // Thread related.
  public final void notify() { ... }
  public final void notifyAll() { ... }
  public final void wait() throws InterruptedException { ... }
  public final void wait(long timeout)
    throws InterruptedException { ... }
  public final void wait(long timeout, int nanos)
    throws InterruptedException { ... }
  // Garbage collector related.
  protected void finalize() throws Throwable { ... }
  // Reflection related.
  public final Class<?> getClass() { ... }
}
```

Dans ce 3ème cours, on va se concentrer principalement sur les 3 premières méthodes.

Exercices

- Est-ce que toutes les classes héritent de `Object` ?
- Ajoutez à `Weapon` les méthodes `clone`, `equals` et `toString`.

```
protected Object clone() throws CloneNotSupportedException { ... }
public boolean equals(Object obj) { ... }
public String toString() { ... }
```

- 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)` ?

Correction exercices

Est-ce que toutes les classes héritent de `Object` ?

Oui, sauf Object lui-même. Vu qu'il n'y a pas de référence circulaire, la classe en haut de la hiérarchie hérite forcément de `Object`. Le `extends Object` est ajouté implicitement si le programmeur ne l'a pas fait.

Correction exercices

Ajoutez à Weapon les méthodes clone, equals et toString.

```
abstract class Weapon {
  protected double damage;
  // ...

  public Object clone() throws CloneNotSupportedException {
    return new Weapon(damage);
  }
  public boolean equals(Object obj) {
    return ((Weapon)obj).damage == damage;
  }
  public String toString() {
    return "Weapon with damage: " + damage;
  }
}
```

Correction exercices

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

```
Weapon axe = new Weapon(10);
Weapon hammer = new Weapon(20);
Weapon axe_c = (Weapon)axe.clone();
Weapon hammer_c = (Weapon)hammer.clone();
if(axe_c.equals(hammer_c)) {
  System.out.println(axe_c + " and " + hammer_c + " are equals.");
} else {
  System.out.println(axe_c + " and " + hammer_c + " are differents.")
}
```

Correction exercices

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 (voir slide suivante).
- 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`.

```
abstract class Weapon { // ...
  public Weapon clone() {
    return new Weapon(damage);
  }
```

Variance

- La variance permet de préciser la relation de sous-typage entre des types complexes (par exemple des signatures de méthodes).
- Voir `http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29`

Co-variance de signature de méthode

Dans notre cas, les types de retours sont co-variants si celui de la plus petite classe est plus précis que celui de la plus grande classe. Exactement ce qui arrive avec `clone`. Ceci évite bien des *casts* inutiles...

Note sur le *downcast*

Coding style

- Le problème du clonage est qu'il faut réaliser un *downcast* (`Object` vers `Weapon`).
- Les *downcasts* sont à éviter, ou alors il faut vérifier leur validité avec `instanceof` ou `getClass()` (attention ils sont distincts, cf. `http://stackoverflow.com/questions/4989818/instanceof-vs-getclass`).
- Néanmoins, généralement, cela montre que le design de l'application est mauvais car forcer la descente dans une hiérarchie pour *une classe en particulier* montre que nos abstractions ne sont pas assez efficaces.
- On peut néanmoins *visiter* élégamment toutes les sous-classes. On y reviendra dans le cours 6/7 avec le *design pattern* visiteur.

Correction exercices

Que se passe t-il si on fait `weapon.equals(new ArrayList())` ou `weapon.equals(null)` ?

- Une exception `ClassCastException` est lancée *à l'exécution* et nous informe que la classe `ArrayList` ne peut pas être convertie en `Weapon`.
- Avec `null`, une `NullPointerException` est lancée.
- Il faut donc vérifier ces cas.

```
public boolean equals(Object obj) {
  if (anObject == null) {
    return false;
  } else if (getClass() != anObject.getClass()) {
    return false;
  } else {
    return ((Weapon)obj).damage == damage;
  }
}
```

Correction exercices

Pourquoi ne pas utiliser `public boolean equals(Weapon w)` ?

Car cette méthode n'est pas *override-equivalent* avec celle d'`Object` et donc le mécanisme d'*overriding* ne fonctionnera pas.

Conclusion

- On a vu le polymorphisme de surcharge et par sous-typage.
- Le premier (*overloading*) est résolu à la compilation.
- Le deuxième (*overriding*) est résolu à l'exécution.
- La classe `Object` fait usage de ces deux principes et partage des traitements communs à tous les objets.
- Veuillez à bien respecter les *coding style* présentés tout au long du cours dans les TPs.