



# L'héritage

Algorithmique et langage de programmation

Gaël Thomas

# Petits rappels : l'objet

- Une structure de données (tuple ou tableau) s'appelle un **objet**
  - Un **objet** possède un **type** appelé **sa classe**
  - Si la classe de l'objet `o` est `C`, on dit que **`o` est une instance de `C`**
- La classe d'un objet définit des
  - **Champs**
  - **Méthodes d'instances**
    - Possède un paramètre implicite du type de la classe nommé **`this`**
  - **Constructeurs** : méthodes spéciales appelées après l'allocation
- Chacun de ces éléments a une visibilité
  - `public` (partout), `private` (local), pas de mot clé (package)

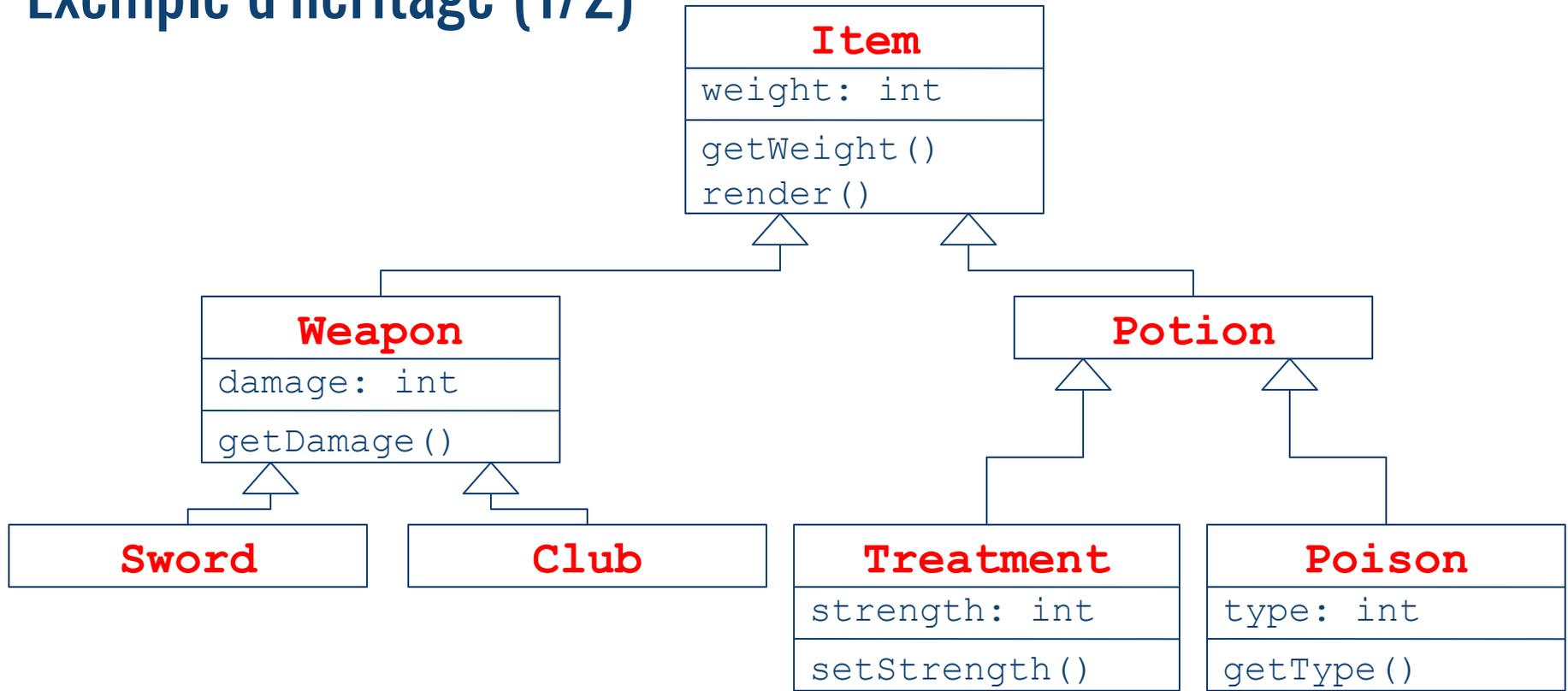
# Principe de l'héritage

- Une classe dite fille peut hériter d'une autre classe dite mère
  - La classe fille possède les champs et méthodes de la mère
  - Et peut en ajouter de nouveaux pour spécialiser la classe mère
  - La classe fille définit un nouveau type
  - Ce type est compatible avec le type de la mère
- L'héritage est, par définition transitif
  - Si C hérite de B et B hérite de A, alors C hérite de A

# But de l'héritage

- Améliorer la réutilisabilité du code
  - En écrivant du code générique et spécialisable
- Exemples
  - Un objet générique dans un jeu, spécialisable en épée ou tunique
  - Un nœud générique dans un arbre binaire de recherche
  - Un flux générique spécialisable en flux réseau, flux vers un fichier, flux reposant sur un tube
  - ...

# Exemple d'héritage (1/2)



# Exemple d'héritage (2/2)

- Le sac du joueur est constitué d'objets de types `Item`
  - Affichés à l'écran avec la méthode `render()`
  - Le poids du sac est calculé en appelant `getWeight()` sur chaque `Item`⇒ le code de gestion du sac est générique
  
- Le système de combat manipule des objets de type `Weapon`
  - Les dégâts sont connus avec la méthode `getDamage()`
  - Le fait de se battre à l'épée ou la massue ne change rien⇒ le code de gestion des combats est générique

# L'héritage en Java

- Une classe hérite **au plus d'une unique** classe
  - Mot clé `extends` suivi de la classe mère

```
class Item {  
    private int weight;  
    public void setWeight(int w) { weight = w; }  
    public Item() {}  
}
```

```
class Weapon extends Item {  
}
```

# L'héritage en Java

- Une classe hérite **au plus d'une unique** classe
  - Mot clé `extends` suivi de la classe mère

```
class Item {  
    private int weight;  
    public void setWeight(int w) { weight = w; }  
    public Item() {}  
}
```

```
class Weapon extends Item {  
}
```

**Weapon hérite** de **Item**  $\Rightarrow$  possède les champs/méthodes/constructeurs de **Item**

```
Weapon weapon = new Weapon();  
weapon.setWeight(10);
```

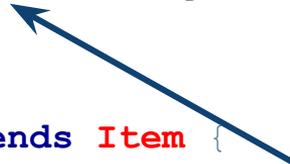
# Héritage et constructeur

- La **première instruction** du constructeur d'une classe fille **doit** être une invocation du constructeur de la classe mère
  - Invocation constructeur mère avec le nom clé `super`

```
class Item {  
    private int weight;  
    public void setWeight(int w) { weight = w; }  
    public Item(int w) {weight = w;}  
}
```

```
class Weapon extends Item {  
    public Weapon(int w) { super(w); }  
}
```

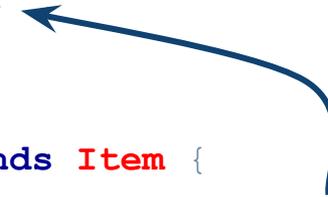
Appel du constructeur  
de **Item** avec le paramètre `w`



# Héritage et constructeur

- La **première instruction** du constructeur d'une classe fille **doit** être une invocation du constructeur de la classe mère
  - Invocation constructeur mère avec le nom clé `super`
  - Invocation implicite si **constructeur mère sans paramètre**

```
class Item {  
    private int weight;  
    public void setWeight(int w) { weight = w; }  
    public Item() {}  
}  
  
class Weapon extends Item {  
    public Weapon(int w) { super(); setWeight(w); }  
}
```



Si `super` omis  
⇒ appel à `super` implicite (ajouté automatiquement à la compilation)

# Héritage et transtypage

- Une fille possède aussi le type de sa mère  $\Rightarrow$  **sur-typage** valide

```
Item item = new Sword(); /* valide (upcast) */
```

- Un objet du type de la mère ne possède pas en général le type de la fille
  - Possibilité de **sous-typer** (*downcast*) un objet explicitement
  - Erreur de transtypage détectée **à l'exécution**

```
Item item = new Sword();
```

```
Sword asSw = (Sword)item; /* valide à l'exécution */
```

```
Club asCl = (Club)item; /* erreur à l'exécution */
```

# Héritage et transtypage

- Un `instanceof` permet de tester si un objet est une instance d'une classe donnée

⇒

- Un 

```
if(item instanceof Club) {  
    Club asCl = (Club)item;  
    System.out.println("This is a club");  
}
```

`Item it`

`Sword asSw = (Sword)item; /* valide à l'exécution */`

`Club asCl = (Club)item; /* erreur à l'exécution */`

# Héritage et appel de méthode d'instance

- Si une fille **surdéfinit** une méthode de la mère, l'appel va toujours vers celui de la **filie** (liaison tardive)

```
class Item {  
    public void render() {  
        /* affiche un ? */  
    }  
}
```

```
class Sword extends Weapon {  
    public void render() {  
        /* affiche une épée */  
    }  
}
```

```
class Engine {  
    public void test() {  
        Item i = new Sword();  
        i.render();  
    }  
}
```

Appel `Sword.render()` car `i` est un objet de type effectif **Sword**  
(même si son type statique dans le code source est **Item**)

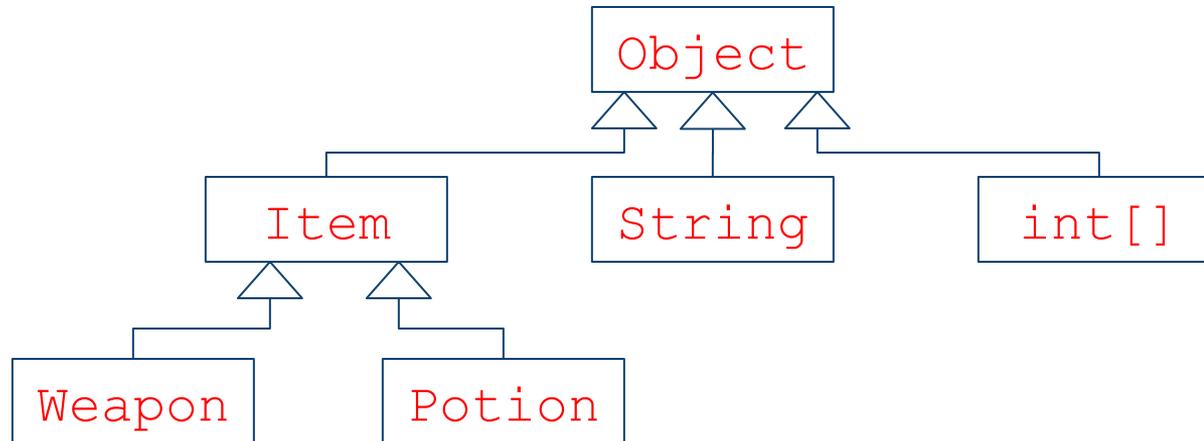
# La visibilité `protected`

- `protected` permet de spécifier qu'une entité (champ ou méthode) est visible des classes filles et du package

# La classe Object

- Toutes les classes héritent d'une classe nommée `Object`
  - Héritage implicite (`extends Object`) si pas de `extends`
  - Héritage par transitivité sinon
  - Vrai aussi pour les classes de tableaux

⇒ La relation d'héritage construit un arbre dont `Object` est la racine



# La méthode `String toString()`

- `Object` fournit quelques méthodes génériques
  - La plus importante à ce stade est `String toString()`
  - Méthode appelée automatiquement pour convertir un `Objet` en `String` quand conversion implicite requise (voir CM1)

```
class City {  
    private String name;  
    public String toString() {  
        return "City: " + name;  
    }  
}
```

```
City city = new City("Paris");  
  
System.out.println(city);  
/* ⇒ affiche City: Paris */
```

# La méthode `String toString()`

- **Object format**

La méthode `String toString()` sert principalement à déterminer les programmes

**Utilisez là !**

Règle générale : toute classe devrait surdéfinir la méthode `String toString()` (par défaut, affiche la référence de l'objet)

# Les classes et méthodes abstraites (1/2)

- Parfois, donner le code d'une méthode dans une classe mère n'a pas de sens
  - La méthode n'est définie que pour être invoquée, c'est aux filles de la mettre en œuvre
  - Exemple typique : la méthode `getDamage()` de `Weapon`

```
class Weapon extends Item {  
    public int getDamage() {  
        /* quelle valeur doit-on renvoyer ici ? */  
    }  
}
```

```
class Sword extends Weapon {  
    public int getDamage() { return 17; }  
}
```

# Les classes et méthodes abstraites (2/2)

- Parfois, donner le code d'une méthode dans une classe mère n'a pas de sens
    - La méthode n'est définie que pour être invoquée, c'est aux filles de la mettre en œuvre
    - Exemple typique : la méthode `getWeight()` de `Item`
  - Dans ce cas, on peut **omettre** le code d'une méthode
    - Marquer la méthode d'instance comme `abstract`
    - Marquer la classe comme `abstract` ( $\Rightarrow$  impossible à instancier)
- $\Rightarrow$  oblige les descendantes concrètes à mettre en œuvre les méthodes marquées `abstract` chez la mère

# Utilisation des classes abstraites (1/3)

// classes abstraites versus classes concrètes

```
abstract class Item {  
    public void printWeight() {  
        System.out.println("Weight: " + getWeight());  
    }  
    public abstract int getWeight();  
}
```

**Club** et **Sword** mettent en œuvre  
getWeight

(on suppose que Weapon hérite de Item)

```
class Club extends Weapon {  
    public int getWeight() {  
        return 42;  
    }  
}
```

```
class Sword extends Weapon {  
    public int getWeight() {  
        return 17;  
    }  
}
```

# Utili

**Remarque** : comme `Weapon` ne met pas elle-même en œuvre `getWeight()`, `Weapon` est une classe abstraite :

```
abstract class Weapon extends Item { }
```

```
// classe
```

```
abstract class Item {  
    public void printWeight() {  
        System.out.println("Weight: " + getWeight());  
    }  
    public abstract int getWeight();  
}
```

`Club` et `Sword` mettent en œuvre

`getWeight`

(on suppose que `Weapon` hérite de `Item`)

```
class Club extends Weapon {  
    public int getWeight() {  
        return 42;  
    } }  
}
```

```
class Sword extends Weapon {  
    public int getWeight() {  
        return 17;  
    } }  
}
```

# Utilisation des classes abstraites (2/3)

```
class Bag {
    Item[] items;

    // exemple d'initialisation de items
    Bag() {
        items = new Item[2];    // tableau de réfs vers Item
        items[0] = new Club();  // ok car Club est un Item
        items[1] = new Sword(); // ok car Sword est un Item
    }
    ...
}
```

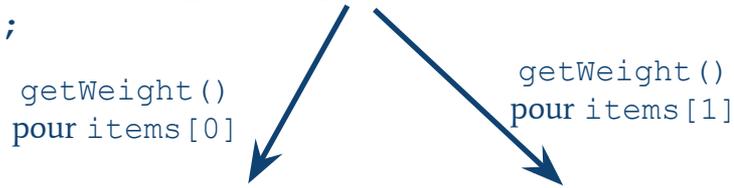
```
class Club extends Weapon {
    public int getWeight() {
        return 42;
    }
}
```

```
class Sword extends Weapon {
    public int getWeight() {
        return 17;
    }
}
```

# Utilisation des classes abstraites (2/3)

```
class Bag {  
    ... // items vaut toujours { new Club(), new Sword() }  
  
    // exemple d'utilisation de getWeight  
    int bagWeight() {  
        int tot = 0;  
        for(int i=0; i<items.length; i++)  
            tot += items[i].getWeight();  
        return tot;  
    }  
}
```

⇒ tot vaut 59  
à la fin de la boucle



```
class Club extends Weapon {  
    public int getWeight() {  
        return 42;  
    }  
}
```

```
class Sword extends Weapon {  
    public int getWeight() {  
        return 17;  
    }  
}
```

# L'interface

- Limitation de l'héritage Java
  - Une classe hérite **au plus** d'une unique classe
  - Que faire, par exemple, pour une classe `Poison` qui serait à la fois une arme (`Weapon`) et une potion (`Potion`) ?
- Solution : l'interface = généralisation des classes abstraites
  - Ne définit **que** des méthodes abstraites, pas de champs
  - Mot clé `interface` au lieu de `class`, plus besoin de marquer les méthodes `abstract`
  - Possibilité pour un objet d'exposer **plusieurs** interfaces
  - On dit alors que la classe **met en œuvre** les interfaces

# L'interface – exemple (1/2)

```
interface Weapon {  
    public int getDamage();  
}
```

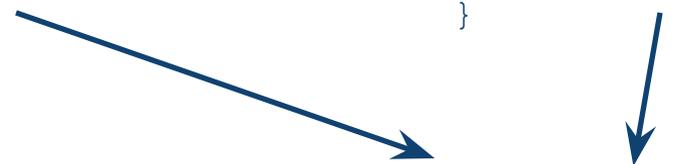
A peu près équivalent à

```
abstract class Weapon {  
    abstract public int getDamage();  
}
```

# L'interface – exemple (2/2)

```
interface Weapon {  
    public int getDamage();  
}
```

```
interface Potion {  
    public void drink(Player p);  
}
```



```
class Poison implements Weapon, Potion {  
    public int getDamage() { ... }  
    public void drink(Player player) { ... }  
}
```

Utilisation :

```
Potion p = new Poison();  
p.drink(sauron);
```

# Notions clés

- Si `Y` est fille de `X`
  - `Y` possède les champs et méthodes de `X`, `Y` est aussi du type `X`
  - Déclaration avec `class Y extends X { ... }`
  - Appel du constructeur parent avec le mot clé `super`
  - Le receveur d'un appel de méthode est donné par le type effectif
- Méthode `abstract`
  - Méthode d'instance sans corps
  - La classe doit être marquée `abstract`, elle est non instanciable
- Interface = classe avec uniquement des méthodes `abstract`
  - Mise en œuvre d'une interface avec `implements`