Contenu du cours

1. Introduction à la POO
2. Une **classe** : définition de nouveaux objets
3. **Instanciation** et utilisation d'objets
4. Création des objets : les **constructeurs**
5. **Références**, visibilité des variables
6. **Encapsulation** et masquage des données
7. **Statique**, ou d'instance ?
8. **Héritage**
9. **Polymorphisme**
10. **Classes abstraites** et **interfaces**
11. **Introduction aux types génériques**
12. **Exceptions en java**
13. *Compléments syntaxiques*

Polymorphisme

Regroupement d'objets, transtypage et liaison dynamique (late binding)

Le concept de polymorphisme en POO

- **Polymorphisme ?**
  - **Grec « poly » = plusieurs**
  - **Grec ancien μορφή « morfê » = forme**

Le **polymorphisme** est la capacité d'un objet à exister sous plusieurs formes (= types) : son type réel, ou celui de sa super-classe.

- **En pratique**

Le **polymorphisme** est vu comme la capacité de choisir dynamiquement (= à l'exécution) la méthode correspondant au type réel de l'objet.

- **Exemple :**

```
HorairePrecis hp = new HorairePrecis();
Horaire h = hp;
Ecran.afficher(h.toString());
```

Typage de *h* : OK
hp « est un » horaire

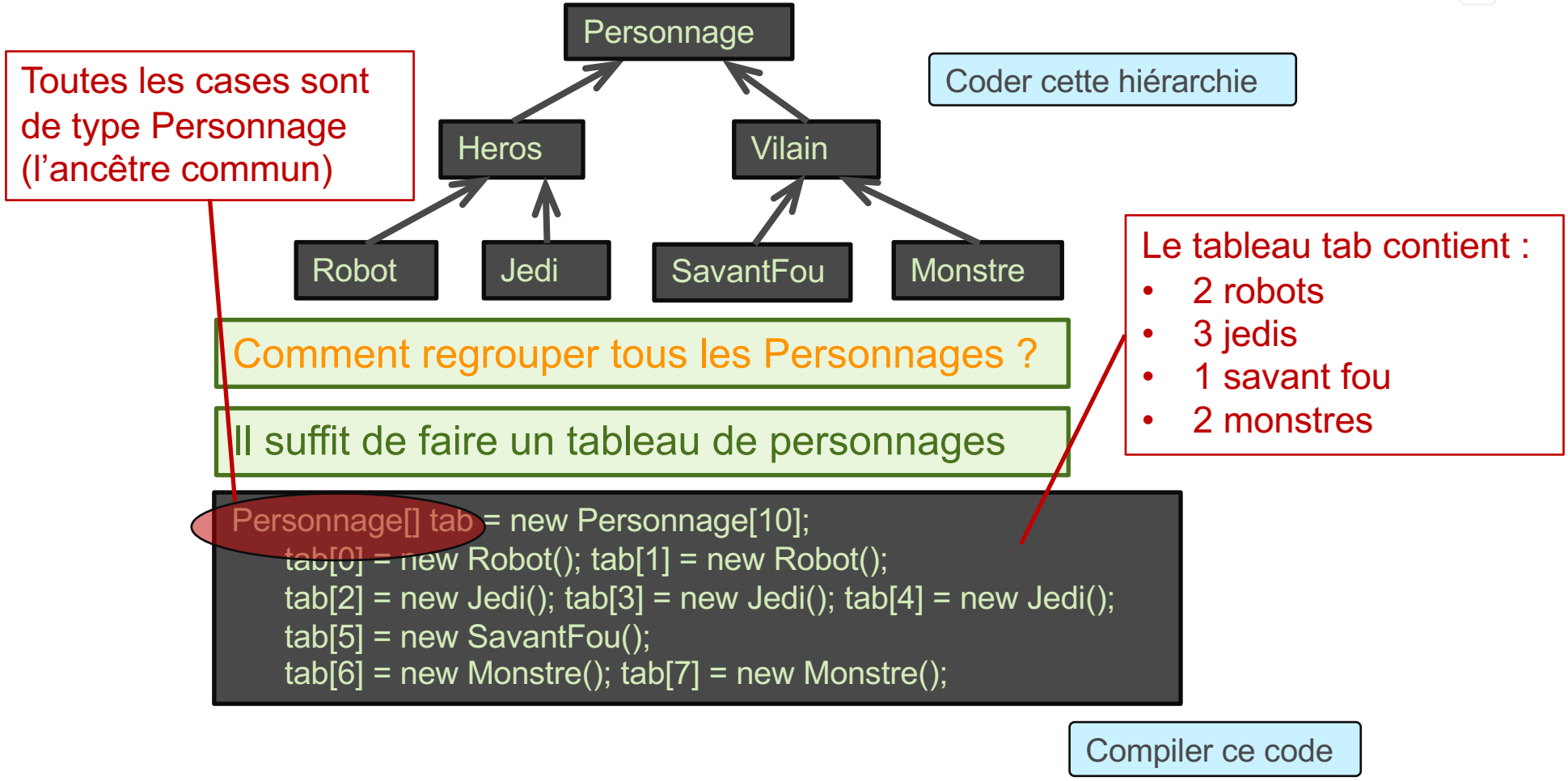
h s'affichera comme un **horaire précis**

  - **Typage statique : h est du type Horaire**
  - **Typage dynamique (à l'exécution) : h se comporte en HorairePrécis**

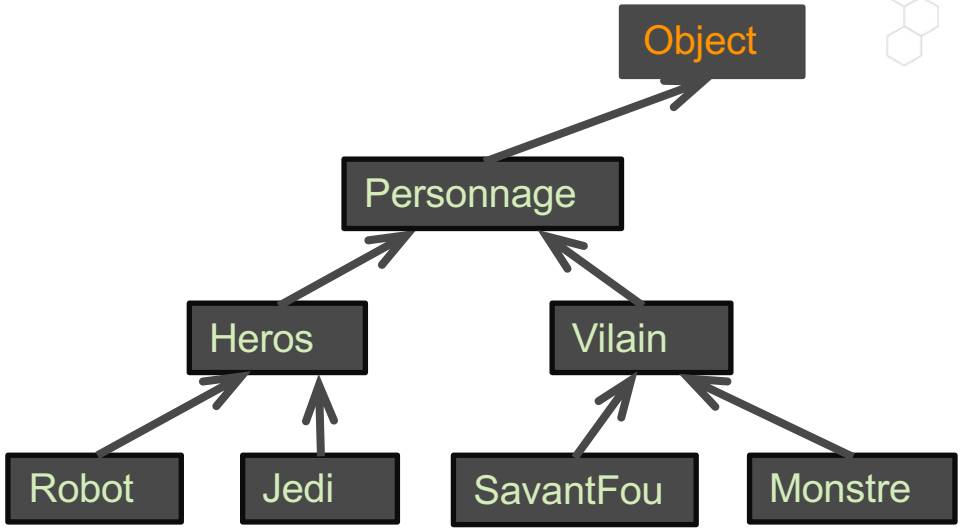
Montrer

Regroupement d'objets polymorphes par héritage (1)

- **Exemple : Types de personnages d'un jeu**



Regroupement d'objets polymorphes par héritage (2)



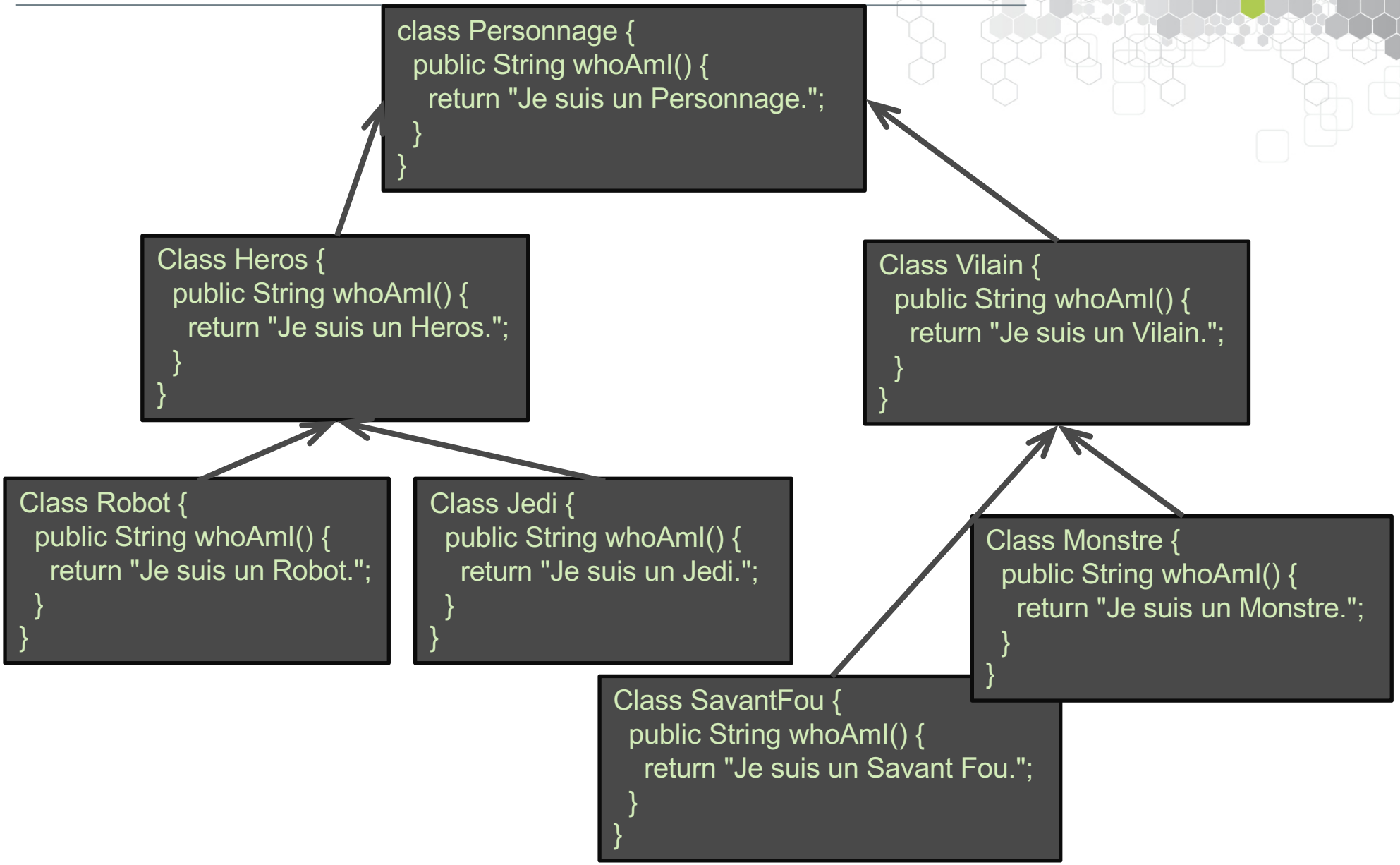
- **On peut aussi séparer les héros et les vilains**

```
Heros[] tabHeros  = new Heros[50];
Vilain[] tabVilains = new Vilain[50];
```

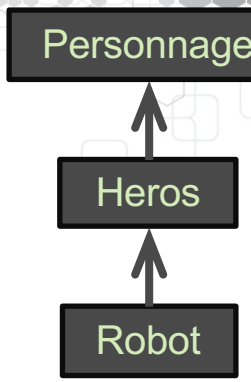
En généralisant, un tableau de Object peut contenir n'importe quel type d'objet.

```
Objet[] tab = new Object[100];
tab[0] = new NombreComplexe();
tab[1] = new HorairePrecis();
tab[2] = new SavantFou();
tab[3] = new Jedi();
…
```

Code Java pour l'exemple



Transtypage (*cast*) ascendant : pas besoin de « caster »



- **Transtypage ascendant**
  - **Un robot « est un » personnage**
  - **==> on peut transtyper un Robot en Personnage**

```
Personnage p  = new Robot();
```

  - **Ce transtypage est ascendant : de la sous-classe vers la super-classe (même si elle est indirecte)**

Transtypage ascendant :
- Autorisé à la compilation (pas besoin de *caster* explicitement)
- Valide à l'exécution

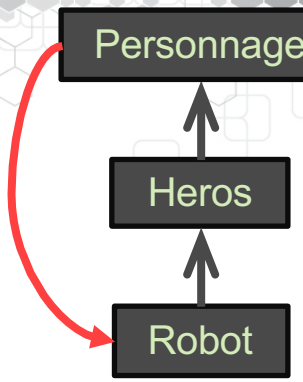
- **Similarité avec les transtypages de primitifs**

```
int n = 5;
double p  = n;
```

OK car un entier est aussi un réel

byte ⊂ short
short ⊂ int
char ⊂ int
int ⊂ long
long ⊂ float
float ⊂ double

Transtypage descendant : il faut « caster »



- **Transtypage descendant**
  - De la super-classe vers la sous-classe
  - N'est pas forcément valide

Un Personnage n'est pas toujours un Robot.

```
Personnage[] tabPersos = new Personnage[20];
tabPersos[0] = new Robot();
...
Robot rob = tabPersos[0];
Robot rob = (Robot) tabPersos[0];
```

OK : transtypage ascendant

KO : transtypage descendant

OK : cast explicite

Transtypage descendant :
- Autorisé uniquement d'une super-classe vers une sous-classe
- Doit être casté explicitement pour la compilation
- Attention : Validité à l'exécution non garantie

```
Jedi jed = (Jedi) tabPersos[0];
```

Compilation : OK (grâce au cast)

Exécution : KO (tabPersos[0] est un Robot, pas un Jedi)

Montrer

Transtypage descendant (= *cast*) : à éviter

- **Quand utiliser le transtypage descendant ?**

Quand on a transformé une instance dans un type trop général (ex: Object)

```
Object o = new Robot();
Ecran.afficher( ((Robot) o).whoAmI() );
```

Indispensable pour compiler

cf. CM/TP sur Génériques

- **Avant l'introduction des « génériques »**
  - **Méthodes génériques : instances et paramètres Object**
  - **Nécessité de caster pour les invoquer**
- **Depuis l'introduction des génériques (≥ Java 1.5)**

Caster ? Le moins souvent possible !

  - **Hors cas particuliers, peut révéler 1 défaut de modélisation**
  - **Souvent utilisé « pour que ça marche »**
  - **Utiliser de préférence les classes et méthodes abstraites, et les types génériques**

cf. chapitre Classes Abstraites

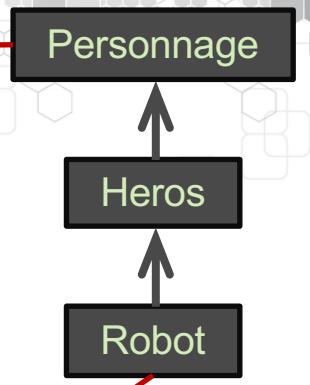
Polymorphisme : liaison dynamique (*late binding*)

- **Exemple de polymorphisme**

```
Personnage p = new Robot();
Ecran.afficher( p.whoAmI() );
```

Méthode définie dans Personnage

Compilation



Question : Quel est le type de p ?

Réponse : Ça dépend !

Méthode *redéfinie* dans Robot

Exécution

- **Typage statique (= à la compilation)**
  - **La variable p est de type Personnage**
  - **Seules les méthodes de Personnage lui sont applicables**
- **Typage dynamique (= à l'exécution)**
  - **A l'exécution, p continue d'être un Robot**

```
Ecran.afficher( p.whoAmI() );
```

Affiche : « Je suis un robot. » !

  - **La version de la méthode à utiliser est déterminée à l'exécution : c'est celle du type réel**