

Héritage

Karima Boudaoud
IUT- R&T



Héritage et Polymorphisme

- Héritage
 - Au cœur de la programmation orientée objet
- Polymorphisme
 - À tes souhaits !



© 1999 Peter Sander

2

Programmation Orientée Objet Réutilisabilité

- Par composition
 - Réutilisation de l'implémentation
 - Objet *c* a un objet *d*
 - *c* peut utiliser les méthodes de *d*
- Par héritage
 - Réutilisation de l'interface
 - Objet *B* est aussi un objet *A*
 - *B* fait tout ce que fait *A* (et peut-être plus)

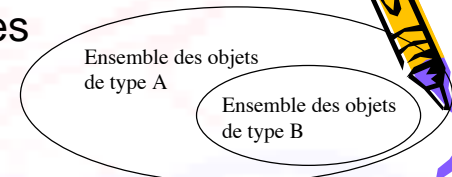


© 1999-2002 Peter Sander

3

Héritage

- Relation entre classes
- Idée clé : « est un »
- B « est un » A
 - Un objet de type B est aussi un objet de type A
 - L'ensemble des objets de type B est contenu dans l'ensemble des objets de type A
 - Tout ce qui est vrai pour A est vrai pour B



* Ce qui est vrai pour B ne l'est pas forcément pour A



© 1999-2002 Peter Sander

4

Héritage

B « est un » A

- Appartenance
 - Un objet de type B est aussi un objet de type A
 - Un objet de type A n'est pas forcément un objet de type B
 - L'ensemble des A est plus grand
- Comportement
 - Tout ce que sait faire un objet de type A, un objet de type B sait le faire aussi
 - Un objet de type B peut savoir faire plus de choses qu'un objet de type A
 - Les B sont plus doués



© 1999 Peter Sander

5

Héritage

- Se dit aussi...
 - B hérite de A
 - B spécialise A
 - A généralise B
 - B est dérivée de A
 - B est une sous-classe de A
 - A est la super-classe de B
 - A est la classe de base de B
 - ...



© 1999 Peter Sander

6

Héritage

- Classe **Clock**
 - Afficher une horloge
- Classe **WorldClock**
 - Afficher l'heure à une ville donnée
- Faut-il tout recommencer ?



© 1999 Peter Sander

7

Héritage

- B hérite de A
 1. Un B sait faire tout ce qu'un A sait faire
 - B acquiert le comportement (l'interface) de A
 - B peut l'utiliser tel quel
 - B peut le modifier en changeant l'implémentation
 2. Un B peut savoir faire plus qu'un A
 - B ajout son comportement propre
 - B est une spécialisation de A

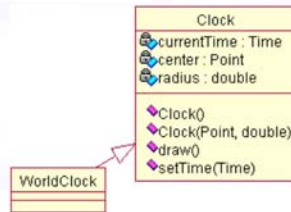


© 1999-2002 Peter Sander

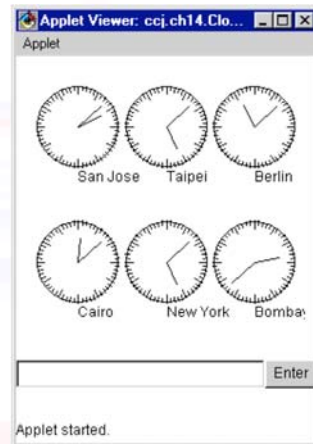
8

Héritage

- Une horloge avec ville **est une** horloge



```
public class WorldClock
extends Clock {
```



9

© 1999 Peter Sander

Héritage

- WorldClock hérite de Clock
- Tout ce qui est vrai pour Clock est vrai pour WorldClock

– E.g.,

```
...
WorldClock wc = new WorldClock(...);
wc.draw();
```

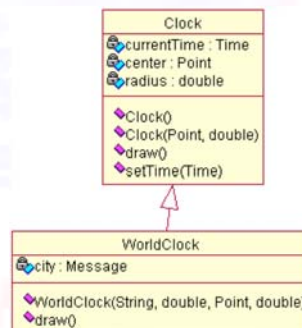
– WorldClock hérite la méthode public draw ne la modifie pas

10

© 1999 Peter Sander

Héritage Spécialisation

- B hérite de A \Leftrightarrow B spécialise A
 - B ajoute du nouveau comportement à A



11

© 1999 Peter Sander

Héritage Spécialisation

- Constructeur

```
public WorldClock(String cityName, double
hourDiff, Point center, double
radius) {
    super(center, radius); // construit super-classe
    // positionne le nom
    Point p = (Point) center.clone(); // v. locale
    p.move(0, -radius);
    // initialisation spécialisée
    city = new Message(p, cityName); // v. d'instance
    Time now = new Time(); // v. locale
    now.addSeconds(hourDiff * SECONDS_PER_HOUR);
    setTime(now);
}
```

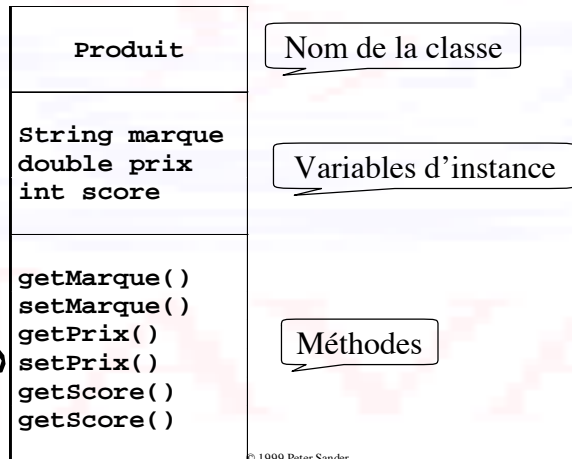
12

© 1999 Peter Sander

Héritage

Représentation Schématique

- D'une classe



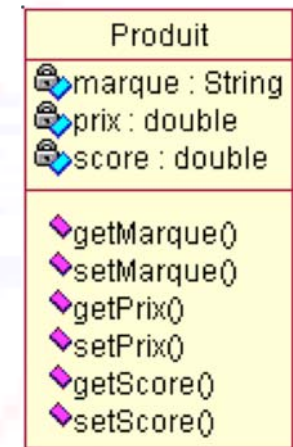
13

© 1999 Peter Sander

Classes et Objets

Représentation en UML

- Présentation UML
 - Unified Modeling Language
 - Représentation standardisée
- Représentation schématisée
 - d'une classe
 - ses variables
 - ses méthodes
 - les rapports entre classes...



14

© 1999 Peter Sander

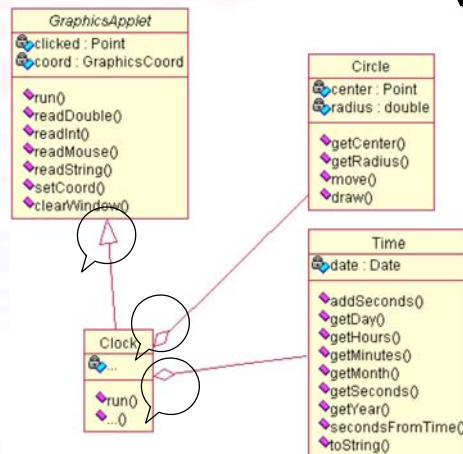
Classes et Objets

Représentation en UML

Clock **spécialise** GraphicsApplet

Clock **utilise un** Circle

Clock **utilise un** Time



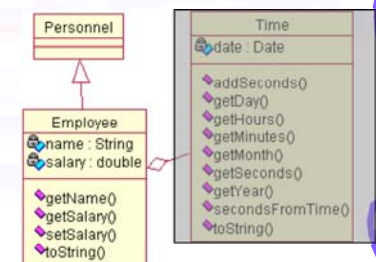
15

© 1999 Peter Sander

Représentation

Spécialisation

- « Spécialise » \Leftrightarrow « est un »
 - Employee spécialise Personnel
 - Employee est un Personne
 - Peut-être avec des choses en plus
 - Indiqué par



16

© 1999 Peter Sander

Classes et Objets Spécialisation

- Une classe spécialise forcément une autre classe
 - Exemple : `Clock` spécialise `GraphicsApplet`

```
public class Clock extends GraphicsApplet {  
    ...  
}
```



© 1999 Peter Sander

17

Classes et Objets Spécialisation

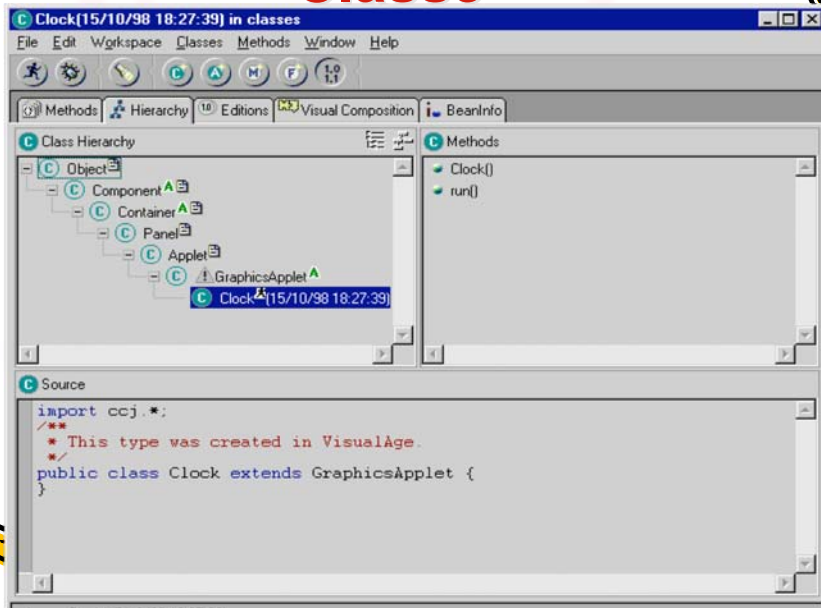
- Hiérarchie de spécialisation
 - `A` spécialise `B` qui spécialise `C`...
...qui spécialise `java.lang.Object`
 - Toute classe spécialise éventuellement `Object`
 - `Object` est la *superclasse cosmique* de Java
 - n'importe quel objet sait faire ce qu'un `Object` sait faire



© 1999-2002 Peter Sander

18

Classes et Objets Classe



19

Classes et Objets Spécialisation

- Une classe spécialise forcément une autre classe
 - Spécialisation implicite de `java.lang.Object`

```
public class Toto {  
    ...  
}
```

équivalent à

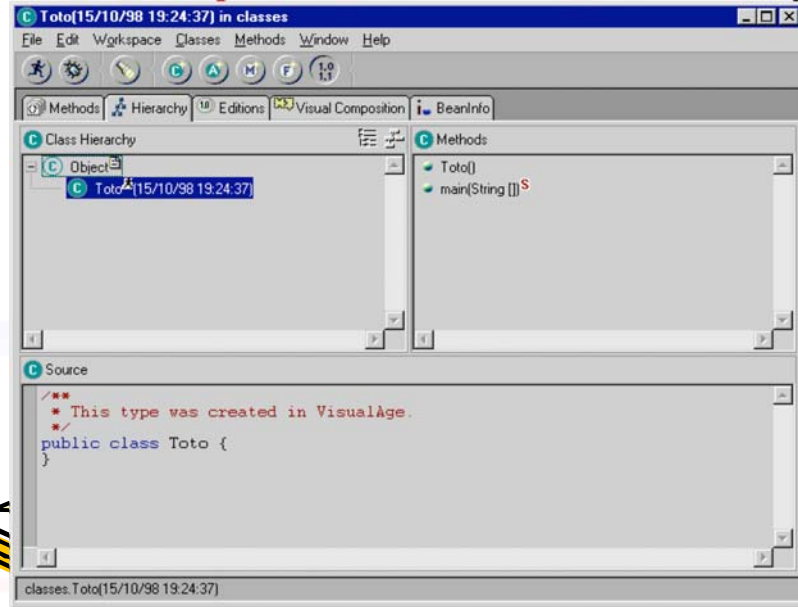
```
public class Toto extends java.lang.Object {  
    ...  
}
```



© 1999 Peter Sander

20

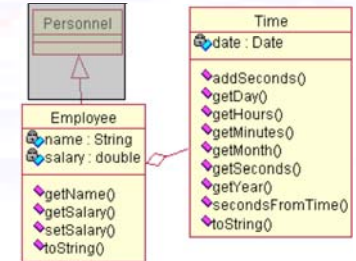
Classes et Objets Spécialisation



21

Représentation Composition

- « Utilise » ⇔ « a un »
 - Ou encore « comprend un » ou « contient un »
 - **Employee a un Time** (date d'embauche)
 - Indiqué par
 - On parle d'« agrégation »
 - composition
 - délégation



22

Classes et Objets Composition

- **Clock** utilise (a, comprend, contient)
 - un **Circle**
 - un **Time**

```

public class Clock extends
GraphicsApplet {
    private Circle clockFace;
    private Time time;
    ...
}

```

- Variables d'instance en accès **private**

23

Héritage

- B hérite de A
 1. Un B sait faire tout ce qu'un A sait faire
 - B acquiert le comportement (l'interface) de A
 - B peut l'utiliser tel quel
 - B peut le modifier en changeant l'implémentation
 - B redéfinit une méthode de A
 2. Un B peut savoir faire plus qu'un A
 - B ajout son comportement propre
 - B est une spécialisation de A
 - B peut ajouter une nouvelle méthode
 - B peut surcharger une méthode existante

24

Héritage Redéfinition et Surcharge

- Redéfinition
 - Modifie l'implémentation
 - Laisse l'interface inchangée
- Surcharge (*overloading*)
 - Modifie l'interface
 - Ajoute un comportement



© 1999-2002 Peter Sander

25

Surcharge et Redéfinition Redéfinition

- Redéfinition de la méthode **draw**
 - Définie dans **Clock...**

```
public void draw() {
    new Circle(center, radius).draw();
    int i;
    ...
    drawHand(...);
    drawHand(...);
}
```

– ...redéfinie dans **WorldClock**

```
public void draw() {
    super.draw(); // méthode de sa super-classe
    city.draw();
}
```

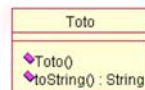
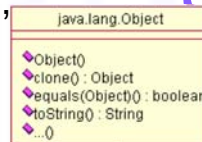


© 1999 Peter Sander

26

Surcharge et Redéfinition Redéfinition

- Dans une classe dérivée, une méthode avec...
 - même signature, même type de retour, que dans une de ses classes de base
 - **toString** de **Toto** redéfinit **toString** de **java.lang.Object**



© 1999-2002 Peter Sander

27

Surcharge et Redéfinition Redéfinition

```
public class Toto extends Titi {
    public String toString() { // redéfinition
        return "Toto#toString here";
    }
    public void strings() {
        System.out.println(this.toString());
        System.out.println(super.toString());
    }
}

public class Titi {}
```



© 1999-2002 Peter Sander

28

Surcharge et Redéfinition

Redéfinition

- Dans **Toto**

```
public void strings() {  
    System.out.println(this.toString());  
    System.out.println(super.toString());  
}
```

Invoke la méthode redéfinie
->Toto#toString here

Invoke la méthode de la super-classe de **Toto**...
...mais **Titi** n'a pas redéfinie **toString**...
...donc c'est la méthode de sa super-classe...
... qui est **java.lang.Object**
...essi.sander.hax.Toto@6353

29

© 1999-2002 Peter Sander

Surcharge et Redéfinition

Redéfinition

- On peut invoquer une méthode de la super-classe...

```
super.uneMethode(...)  
- ...mais pas de la super-super-classe  
super.super.uneAutreMethode(...)  
// non !
```

30

© 1999 Peter Sander

Surcharge et Redéfinition

Surcharge

- Dans une classe, deux méthodes avec...
 - le même nom
 - des signatures différents

```
Clock()  
Clock(Point, double)
```

- Compilateur en choisira la bonne
 - en fonction des arguments

31

© 1999 Peter Sander

Surcharge et Redéfinition

Surcharge

- Compilateur interdit deux méthodes avec
 - la même signature
 - des valeurs de retour différentes

```
int uneMethode(String, int, double)  
double uneMethode(String, int, double)
```

– Compilateur ne saura pas comment choisir

```
...  
uneMethode("coucou", 7, 3.14) // laquelle?
```

32

© 1999 Peter Sander

Héritage Niveau d'Accès

- **public**
 - Accessible à tous
- **private**
 - Accessible que dans la classe ou c'est déclaré
 - N'est même pas visible depuis l'extérieur
 - erreur de compilation



© 1999 Peter Sander

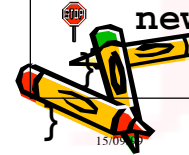
33

Héritage Niveau d'Accès

```
public class Titi {  
    private void uneMethodePrivéeDeTiti()  
        System.out.println("Hello !");  
    }  
}
```

– Erreur de compilation

```
public class Toto {  
    public void uneMethode() {  
        new Titi().uneMethodePrivéeDeTiti();  
    }  
}
```



© 1999 Peter Sander

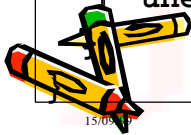
34

Héritage Niveau d'Accès

```
public class Titi {  
    private void uneMethodePrivéeDeTiti() {  
        System.out.println("Hello !");  
    }  
}
```

– Erreur de compilation, même pour classe dérivée

```
public class Toto extends Titi {  
    public void uneMethode() {  
        uneMethodePrivéeDeTiti();  
    }  
}
```



© 1999 Peter Sander

35

Héritage Niveau d'Accès

- Le redéfinition ne peut réduire le niveau d'accès

```
public class Titi {  
    public void uneMethode() {...}  
}
```

– Erreur de compilation

```
public class Toto extends Titi {  
    private void uneMethode() {...}  
}
```



© 1999 Peter Sander

36

Héritage

- L'héritage permet de recycler du code
- Valable que dans la relation « est un »
- N'en abusez pas
 - Une voiture est un véhicule
 - Un vélo est un véhicule
 - Un vélo n'est pas une roue
 - un vélo contient une roue (parfois deux)
 - Un **WorldClock** est un **Clock**
 - Un **Clock** n'est pas un **Circle**
 - un **Clock** contient un **Circle**
 - Un **Clock** n'est pas un **Time**



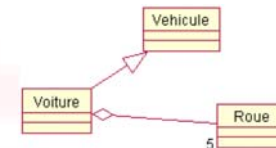
© 1999 Peter Sander

37

Composition

- Pour la relation « contient un »...
...préférez la composition

```
public class Voiture extends Vehicule {
    private Roue[] roue = new Roue[5];
    ...
}
```



© 1999 Peter Sander

38

Héritage et Composition

- Il n'est pas toujours évident lequel utiliser
- Héritage
 - B hérite de A si
 - B « est un » A
 - peut être utilisé quand toutes les méthodes de A s'appliquent aussi à B
- Composition
 - B contient A si
 - B « utilise un » A
 - quand au moins une méthode de A ne s'applique pas à B

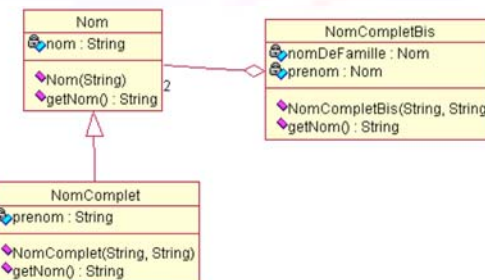


© 1999 Peter Sander

39

Héritage et Composition

- Exemples d'héritage et de composition
 - Même comportement



© 1999 Peter Sander

40

Héritage et Composition

```
/**
 * Stocke un nom.
 */
public class Nom {
    ...
    /**
     * Accède au nom stocké.
     * @return le nom
     */
    public String getNom() {...}
```



41

© 1999 Peter Sander

Héritage et Composition

```
public class Nom { // classe de base
    private String nom; // var. d'instance
    public Nom(String nom) { // construct.
        this.nom = nom;
    }
    public String getNom() { //
        accesseur
        return nom;
    }
}
```

42

© 1999 Peter Sander

Héritage et Composition Héritage

```
/**
 * Stocke un prénom et un nom de
 * famille.
 * Le nom de famille est une variable
 * de la super-classe, le prénom es
 * une variable de cette classe.
 */
public class NomComplet extends Nom {
```

43

© 1999 Peter Sander

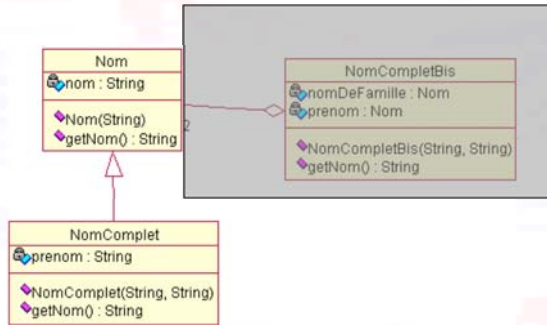
Héritage et Composition Héritage

```
public class NomComplet extends Nom {
    private String prenom; // nouvelle var.
    public NomComplet(String
        nomDeFamille, String prenom) { // constr.
        super(nomDeFamille);
        this.prenom = prenom;
    }
    public String getNom() { // redéfinit.
        return prenom + " " + super.getNom();
    }
}
```

44

© 1999 Peter Sander

Héritage et Composition Héritage



© 1999 Peter Sander

45

Héritage et Composition Composition

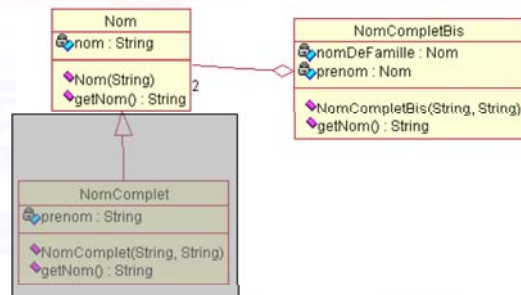
```

/**
 * Stocke un prénom et un nom de famille.
 * Les deux sont des variables de type
 * Nom de cette classe.
 */
public class NomCompletBis {
    ...
}
  
```

© 1999 Peter Sander

46

Héritage et Composition Composition



© 1999 Peter Sander

47

Héritage et Composition Composition

```

public class NomCompletBis {
    private Nom prenom; //une var. d'instance
    private Nom nomDeFamille; // et l'autre
    public NomCompletBis(String nomDeFamille,
        String prenom) { // constructeur
        this.nomDeFamille = new
        Nom(nomDeFamille);
        this.prenom = new Nom(prenom);
    }
    public String getNom() { // accesseur
        return prenom.getNom() + " " +
        nomDeFamille.getNom();
    }
}
  
```

© 1999 Peter Sander

48