R2.01 : cours n°6

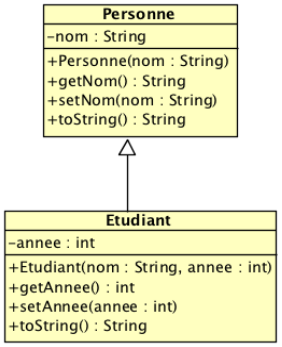
I.Borne

- Héritage (2ème partie)
- Retour les visibilités

Ce que l'on sait pour l'instant

- Toute classe est sous-classe d'une autre classe.
- Par défaut la superclasse est Object
- Une classe peut avoir plusieurs sous-classes mais ne peut avoir qu'une seule superclasse directe
- Chaque classe est responsable de la création/initialisation des ses attributs d'instance dans son constructeur.
- Une classe hérite des attributs et méthodes d'instance de sa superclasse.
- L'héritage permet de surcharge les méthodes de la superclasses

Chaînage des constructeurs : reprise de l'exemple




```
public class Personne {
   private String nom;

   public Personne (String unNom) {
        this.setNom(unNom);
   }
...}
```

Le constructeur de Personne initialise l'attribut *nom*
Quand on crée une instance d'Etudiant on lui donne un nom

Appel du constructeur de la classe mère

```
public class Personne {
   private String nom;

   public Personne (String unNom) {
        this.setNom(unNom);
   }
```

L'appel à super doit être la première instruction

```
public class Etudiant extends Personne {
   private int annee;

    public Etudiant (String unNom, int uneAnnee) {
         super(unNom);  //appel du constructeur de Personne
         this.setAnnee(uneAnnee);
    }
```

Suite Appel du constructeur de la classe mère

```
public class Etudiant extends Personne {
   private int annee;

    public Etudiant (String unNom, int uneAnnee) {
          super();
          this.setAnnee(uneAnnee);
    }
```

Que se passe-t-il ici ?

```
public class Etudiant extends Personne {
   private int annee;

    public Etudiant (String unNom, int uneAnnee) {
          this.setAnnee(uneAnnee);
          super(nom) ;
    }
```

et ici ?

Héritage des constructeurs sans paramètres

```
public class Personne {
   private String nom;

   public Personne() {
        this.setNom("non défini");
   }
}
```

appel par défaut du super constructeur sans paramètre

```
public class Etudiant extends Personne {
   private int annee;

    public Etudiant() {
          super();
          this.setAnnee(1);
    }
}
```

équivalents

```
public Etudiant() {
   this.setAnnee(1);
    }
```

Mais c'est mieux d'être explicite !

Héritage et constructeurs

- **Appel explicite** du constructeur de la superclasse avec passage de paramètres.
- **Appel implicite** du constructeur sans paramètre de la superclasse.
- Dans tous les cas l'appel à super se fait comme première instruction du constructeur.
- Pour un appel explicite il faut respecter l'ordre et le nombre des paramètres du constructeur de la superclasse.

Pseudo-fonction super()

- La pseudo-fonction ou pseudo-méthode **`super()`** est utilisée dans les constructeurs.
- Elle a pour effet d'appeler le constructeur du parent de même signature. Ceci permet de transmettre les arguments de construction des objets.

Utilisation de this()

this() fait référence au constructeur sans argument dans un autre constructeur de la même classe.

Il doit être placé comme la 1ère ligne du constructeur.

```
public class A {
   public A(){
     System.out.print("Hello");
   }
   public A(int i) {
     this();
     System.out.println("Bonjour "+i);
   }
}
```


Pseudo-variable super

La pseudo-variable **super** (comme `this`) fait référence à l'objet receveur courant dans une méthode, mais la recherche de la méthode associée commence dans la superclasse.

classe Personne

```
public String toString() {
    return("Nom : " + this.getNom();
}
```

classe Etudiant

```
public String toString() {
    // on appelle la méthode de la superclasse
    String resultat = super.toString() ;
    resultat = resultat +"Année : " + this.getAnnee());
   return (resultat);
}
```

pseudo-variable

- Que fait le code suivant ?

```
public String toString() {
    // on appelle la méthode de la superclasse
    String resultat = this.toString() ;
    resultat = resultat +"Année : " + this.getAnnee());
   return (resultat);
}
```


Dans quels cas doit-on utiliser super ?

- Appel d'une méthode de la superclasse qui porte le même nom que celle dans laquelle on fait l'appel.
- Si la méthode héritée appelée ne porte pas le même nom alors le *this* est suffisant car il n'y a pas d'ambiguité :
  - Si la méthode n'est pas trouvée dans la classe alors elle est recherchée dans la superclasse.

Héritage de la méthode toString()

Dans la classe Object de Java :

toString

```
public String toString()
```

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Returns:

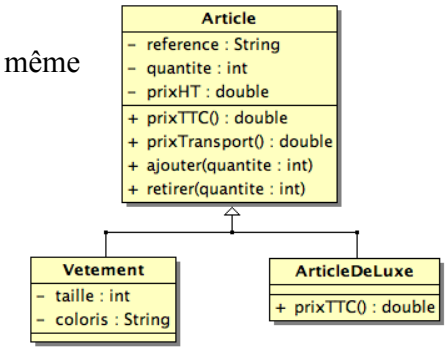
a string representation of the object.

Héritage dynamique et héritage statique

- Héritage statique des attributs d'instance et de classe
- Héritage dynamique des méthodes
- Mécanisme de recherche des méthodes

Héritage statique des variables

- Lors de la définition d'une classe, les attributs des super-classes sont « ajoutés » à ceux qui sont déclarés dans la classe.
- Par conséquent les noms des attributs sont uniques pour une même ligne hiérarchique.



- **Vêtement**
  - **référence (Article)**
  - **quantité (Article)**
  - **prixHT (Article)**
  - **taille**
  - **coloris**

Héritage dynamique des méthodes dans les langages à objets à liaison dynamique

- La recherche d'une méthode ne s'effectue qu'à l'évaluation d'un envoi de message, donc à l'exécution du test.
- Mécanisme de recherche d'une méthode :
  1. Identification de l'objet receveur du message
  2. Recherche de la classe de l'objet receveur
  3. Recherche de la méthode dans la classe de l'objet
  4. Si la méthode est trouvée alors elle est appliquée
  5. Si la méthode n'est pas trouvée dans la classe de l'objet receveur, alors la recherche se poursuit dans les super-classes
  6. Enfin si la méthode n'est pas trouvée dans la classe racine *Object* alors un message d'erreur est envoyé.

Surcharge versus masquage des méthodes

- *surcharge* (ou overloading) : deux méthodes ont le même nom mais des types de paramètres différents donc les deux méthodes ont des signatures différentes.

```
public Personne()
public Personne(String unNom)
```

- *masquage* (ou overriding): deux méthodes ont le même nom et exactement les mêmes des types de paramètres donc les deux méthodes ont des signatures identiques.

Mais dans des classes différentes

```
public String toString()
```

La méthode **`toString()`** de la classe **`Etudiant`** masque la méthode **`toString()`** de la classe **`Personne`**

Mécanisme de recherche des méthodes

- La recherche des méthodes est dynamique.
- Rien n'empêche que certaines méthodes d'une sous-classe aient le même nom que certaines méthodes d'une super-classe.
- L'ambiguïté est levée par l'ordre du parcours : c'est la première méthode rencontrée qui sera adoptée.
- La pratique courante consiste à redéfinir au niveau de la sous-classe une méthode, déjà existante dans la super-classe, afin de l'adapter aux instances de cette sous-classe.
- Exemple :
  - la méthode `toString()` qui est héritée de la classe `Object` est redéfinie dans la plupart des classes.
- On parle de polymorphisme dans ce dernier cas.

Résumé des concepts d'héritage

- **Héritage** : définir une classe comme extension d'une autre
- **Superclasse** : une classe étendue par une autre
- **Sous-classe** : une classe qui hérite de tous les attributs et méthodes d'intance de sa superclasse.
- **Hiérarchies d'héritage** : Les classes liées par des relations d'héritage forment une hiérarchie d'héritage.
- **Constructeurs de superclasses :** Le constructeur d'une sous-classe doit toujours invoquer le constructeur de sa superclasse comme première instruction. Si le code n'inclut pas d'appel de ce type, Java tente d'insérer un appel automatiquement.
- **`Object :`** la superclasse par défaut de toutes les classes sans superclasse explicite.

Retour sur les visibilités

Méthode privées /publiques

Visibilité protected : package/héritage

Retour sur les visibilités public/private/...

- Attribut ou méthode d'une classe déclaré *private* => accessible uniquement à l'intérieur de la classe
- Attribut ou méthode d'une classe déclaré *public* => membre accessible de partout (ouvert au «monde»)
- Si aucune précision de visibilité sur la méthode ou l'attribut, alors par défaut la visibilité est de type ***package*** => accessible uniquement par les classes du même **package**.

Visibilité d'un classe (1) - public

- Une classe a deux visibilités possibles : public et package

```
package bank;
public class CompteBancaire {
    private int numero;
    private double solde;
```

La classe CompteBancaire a une visibilité publique : elle est utilisable par toutes les classes, qu'elles appartiennent ou non au package « bank ».

Visibilité d'un classe (2) - package

```
package bank;
class CompteBancaire {
    private int numero;
    ...
    public int getNumero() {
      return (this.numero);
    }
```

- La classe CompteBancaire n'a pas une visibilité publique, mais package : elle est utilisable uniquement par les classes qui appartiennent au package « bank ».
- Conséquence pour les membres (attributs et méthodes) de la classe :
  - le membre privé reste privé,
  - **restriction** pour la visibilité du membre publique qui devient «**package**».

Visibilité des méthodes

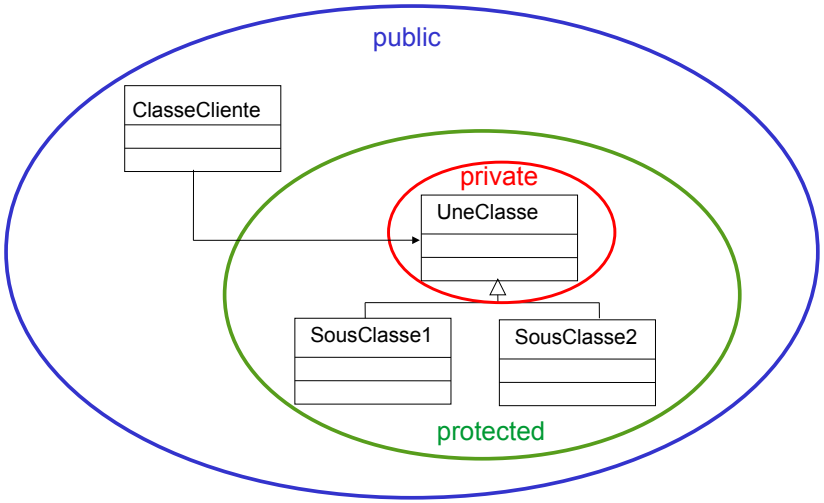
- Pour l'instant nous avons surtout utilisé des méthodes publiques :
  - Visibles à l'extérieur de la classe par tout le monde
- Parfois on définit une méthode dans une classe qui n'a pas besoin d'être visible à l'extérieur de la classe :
  - Elle ne fait pas partie de l'interface publique de la classe
  - Elle fait une tâche utile à la classe, par exemple une initialisation de données ou un calcul particulier.

Une méthode déclarée **private** dans une classe ne sera visible qu'à l'intérieur de la classe.

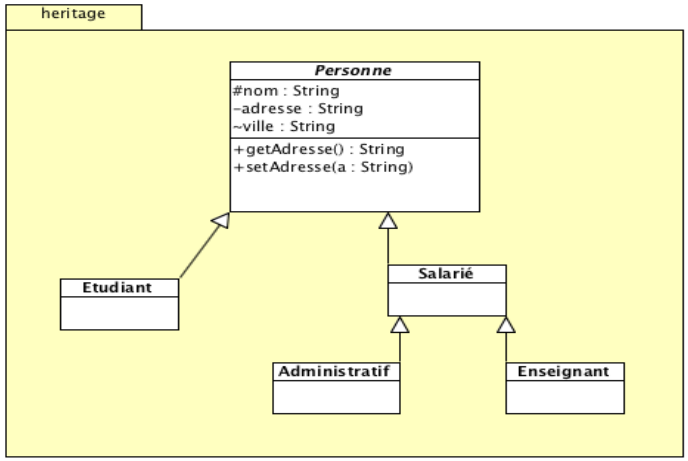
Visibilité avec héritage et package

- Si un attribut ou une méthode est déclaré **`public`**, alors il est accessible par toutes les classes : les classes clientes et toutes les sousclasses.
- Si un attribut ou une méthode est déclaré **`private`** alors il n'est accessible que de sa classe.
- Si un attribut ou une méthode est déclaré **`protected`** alors il est accessible de partout dans le package de la classe A si A est publique, et dans les classes héritant de A même dans d'autres packages.

Visibilité et héritage : 3 niveaux de visibilité



Notation des visibilités en UML



\+
\-
\#
~

Utilisation habituelle de protected

- Dans un monde idéal, les membres private devraient toujours être privés.
- Dans les projets réels, il arrive que l'on veuille cacher quelque chose au monde au sens large et que l'on veuille permettre l'accès pour les membres des sous-classes.
- La visibilité protected est un moyen de le faire.
- Mais la meilleure approche est de
  - laisser les attributs d'instance avec une visibilité private,
  - permettre l'accès pour les héritiers à travers les méthodes déclarées protected.

Portée des déclarations

Méthode ou attribut (sauf constructeur) déclaré	visible dans A	visible dans les sousclasses de A	visible partout
`private` dans A			
`protected` dans A			
`public` dans A			

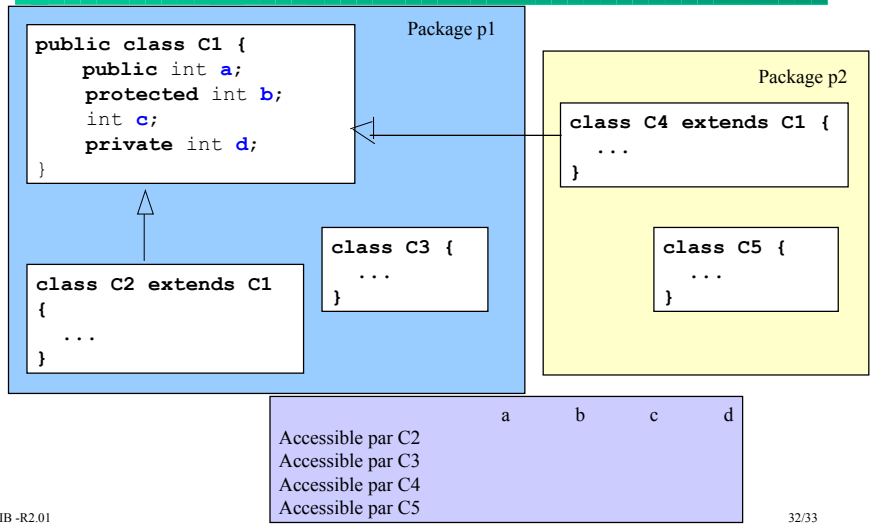
Droits d'accès aux attributs : public, privé ou « de package » (1)

- Un attribut d'instance *privé* peut être accédé uniquement par les méthodes de la classe.
- Un attribut d'instance *public* est accessible via un objet de toutes les méthodes de toutes les classes et de tous les packages. (Attention à la violation de l'encapsulation)
- Un attribut déclaré *protected* est accessible dans son paquetage et dans les classes dérivées (sous-classes).
- Si on ne déclare *rien* l'attribut est accessible dans les méthodes de la classe et dans les méthodes des classes du package.

Droits d'accès aux méthodes : public, privé ou « de paquetage » (2)

- Le principe est le même pour une méthode qui peut être :
  - *privée* et utilisable seulement dans sa classe
  - *public* et utilisable dans toutes les méthodes de toutes les classes
- Par défaut la méthode est accessible dans les classes du package.
- Si une classe n'est pas déclarée *public* (visible uniquement dans son package), alors si ses méthodes sont déclarées *public* elles ne seront néanmoins pas accessibles d'un autre package.

Visibilité des variables et package



	a	b	c	d
Accessible par C2				
Accessible par C3				
Accessible par C4				
Accessible par C5				