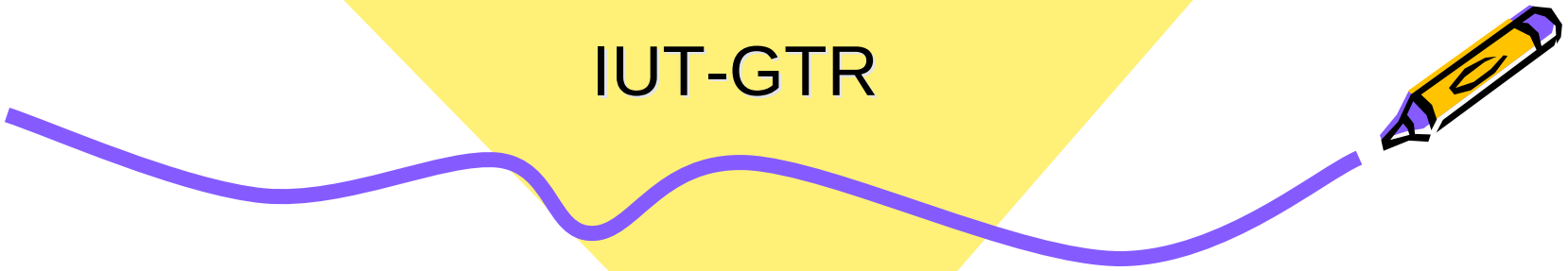
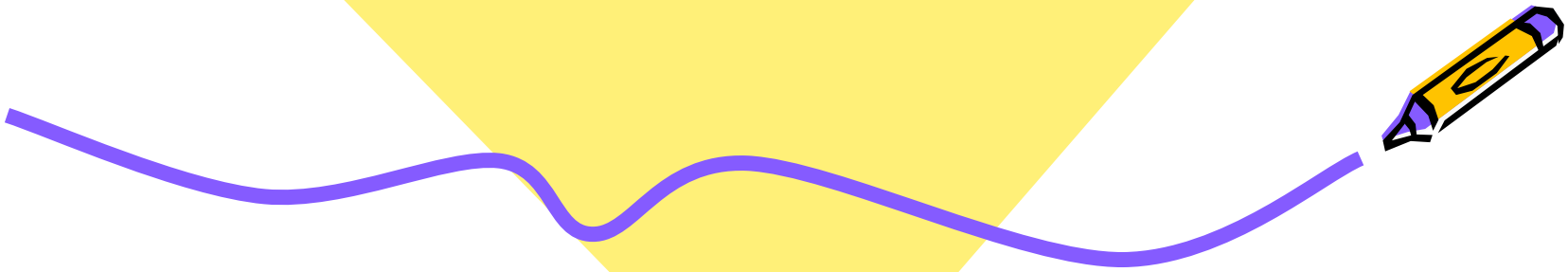
Synthèse I5 (Classes/Objets - Héritage)

Karima Boudaoud

IUT-GTR



Conventions de nommage et recommandations



Conventions de nommage

- les noms doivent être de préférence en anglais (pour faciliter la compréhension du code, dans le cas de développement de logiciels open source)
- les noms doivent être parlants, éviter d'utiliser des abréviations
- les noms des classes et des interfaces doivent commencer par une majuscule et avoir la forme suivante : MyClass
- les noms des attributs static final doivent être en majuscule et avoir la forme suivante s'ils contiennent plusieurs noms : MY_CONSTANT
- les noms des autres attributs et des méthodes doivent commencer par une minuscule et avoir la forme suivante s'ils contiennent plusieurs noms: myAttribut, myMethod()
- les noms de classe héritant de la classe Exception doivent se terminer par Exception: MyException
- les noms de paquetage ne doivent contenir que des minuscules et chaque mot séparé par un « . »

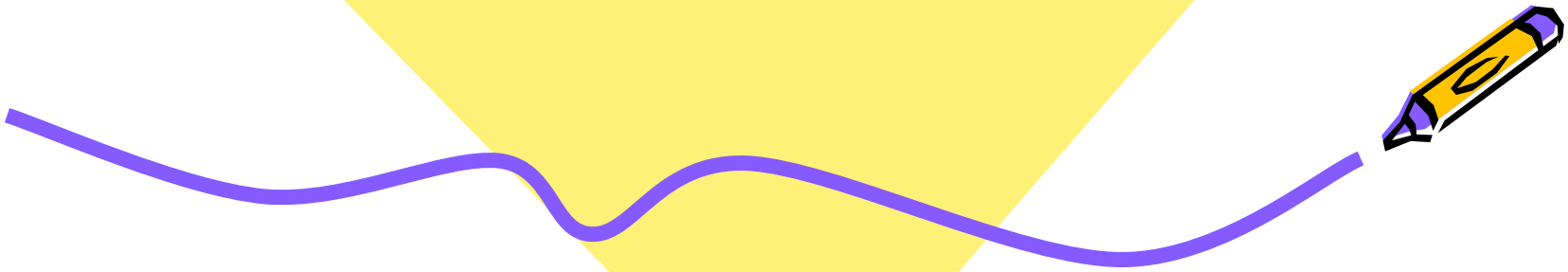
Recommandations (1)

- écrire une méthode main pour chacune des classes testables
- le main de l'application doit être dans une classe séparée Main *(ou Test)*
- éviter de déclarer des attributs comme public. Garder toutes les données private et implantez plutôt pour un attribut « attribut », 2 accesseurs : setAttribut(), getAttribut()
- toutes les données ne nécessitent pas d'accesseurs en modification
  - ✓ la date de naissance d'une personne ne va pas changer
- ne pas utiliser trop de types basiques
  - ✓ remplacer : une rue, une ville, un pays, un code postal par une classe Adresse
- toujours initialiser et explicitez les attributs
- utiliser equals() plutôt que "==" (pour les types non primitifs comme String)

Recommandations (2)

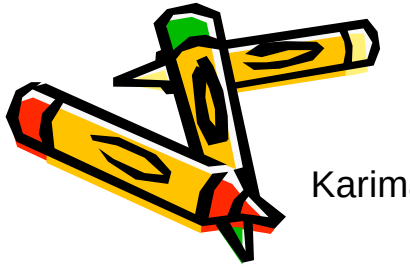
- initialiser les objets qui peuvent retourner une Enumeration(Vector, HashTable,…) avec autre chose que null
- écrire des méthodes qui ne font qu'une chose et en utilisent d'autres
- si possible, écrire un constructeur par défaut pour pouvoir utiliser newInstance()
- ne déclarez les variables locales qu'à l'endroit où vous en avez réellement besoin
- affectez null à toutes les références qui ne sont plus utilisées
- minimisez les static non final
- minimiser les * dans les imports
- utiliser des interfaces aussi souvent que possible si vous pensez que votre implémentation est susceptible d'être remplacée par une autre ultérieurement

Classes et instances

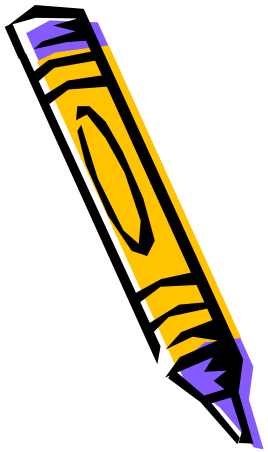


Classes et instances

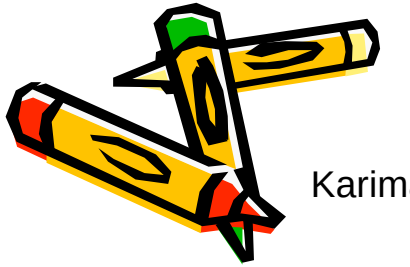
- **Classe**
  - une classe est un moule pour créer des objets
  - une classe permet de modéliser des objets du monde réel
  - c'est un *type d'objet*
  - exemple : une maison, une voiture, un iut,..
- **Objet**
  - un objet est une *instance de classe*
  - exemple : ma maison, ma voiture, l'iut gtr,…



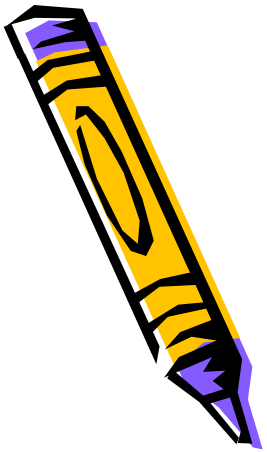
Membres d'une classe



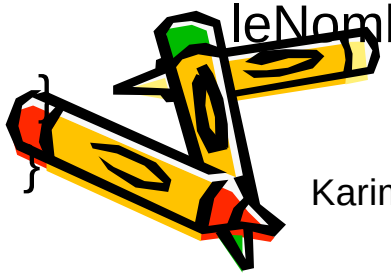
- une classe est composée de :
  - ❒ variables, ou champs
    - ✓qui donnent l'état des instances
  - ❒ constructeurs
    - ✓ qui permettent de créer des nouvelles instances de la classe (objets)
  - ❒ accesseurs et modificateurs
    - ✓ qui permettent de lire et modifier les variables d'instance
  - ❒ méthodes
    - ✓qui indiquent les types de messages qui peuvent être envoyés aux instances



Exemple



```
public class Livre {
    private static final String LE_TITRE =  "inconnu"; // constante
    private static final String L_AUTEUR =  "inconnu"; // constante
    private String leTitre, lAuteur; // variables
    private int leNombrePages;
 public Livre () {// constructeur par défaut
          leTitre = LE_TITRE ; lAuteur = L_AUTEUR ;
 }
 public Livre (String unTitre, String unAuteur) {// constructeur
          leTitre = unTitre; lAuteur = unAuteur;
 }
 public String getLAuteur() { // un accesseur en lecture
          return lAuteur;
 }
 public void setLeNombrePages(int unNombrePages) { // un accesseur en
          leNombrePages = unNombrePages;                modification
 }
}
```



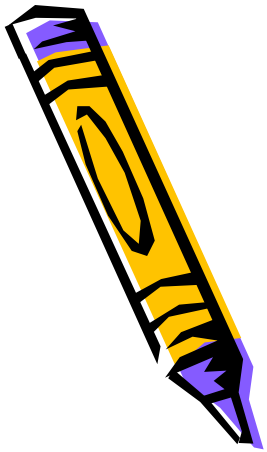
Classe (1)

- pour écrire une classe, il faut respecter la syntaxe suivante :

[*modifiers*] class ClassName [extends SuperClassName]
[implements InterfaceName]

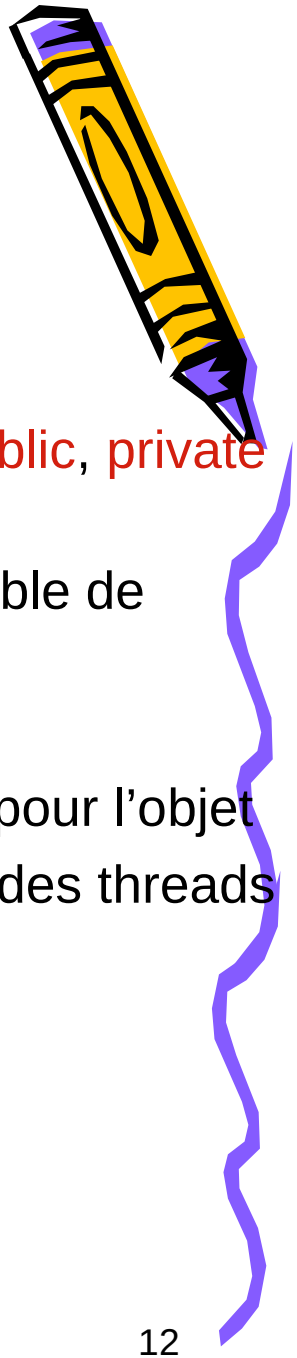
- *modifiers* représente un mot clé qui peut être public, abstract ou final
  - ✓ le mot clé public signifie que la classe peut être utilisée par tous et qu'elle est donc visible de l'extérieur
  - ✓ le mot clé abstract signifie que la classe est composée de méthodes abstraites (i.e des méthodes précédées du mot clé abstract) qui devront être implémentées dans les classes qui hériteront de cette classe
  - ✓ le mot clé final signifie qu'on ne peut pas faire d'héritage avec cette classe (i.e qu'elle ne peut pas être dérivée)

Classe (2)



- l'écriture class ClassName permet de spécifier le nom ClassName de la classe
- l'écriture extends SuperClassName permet de spécifier que la classe ClassName dérive de SuperClassName
- l'écriture implements InterfaceName permet de spécifier que la classe ClassName implémente les méthodes (i.e écrit le code des méthodes) contenues dans InterfaceName

Variable



- pour déclarer une variable, il faut respecter la syntaxe suivante:

[*access*] [static] [final] [transient] [volatile] *Type* variableName

- *access* sert à spécifier la visibilité de la variable, i.e si elle est public, private ou protected (voir plus loin transparent types d'accès)
- le mot clé static permet de spécifier que la variable est une variable de classe
- le mot clé final signifie que la variable est une constante
- le mot clé transient signifie que la variable n'est pas persistante pour l'objet
- le mot clé volatile signifie que la variable peut être modifiée par des threads concurrents et ce de manière asynchrone

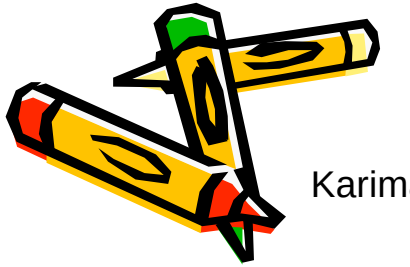
Méthode (1)

- pour écrire une méthode, il faut respecter la syntaxe suivante:

  [*access*] [static] [abstract] [final] [native] [synchronized]
  returnType methodeName([param])
- *access* sert à spécifier la visibilité de la méthode, i.e si elle est public, private ou protected (voir plus loin transparent "Types d'accès")
- le mot clé static signifie que la méthode est une méthode de classe
- le mot clé abstract signifie que la méthode n'est pas implémentée (i.e. qu'elle n'a pas de code). Cette méthode ne sera implémentée que dans la sous-classe qui hérite de la classe où elle (i.e la méthode) est définie. Dans ce cas là elle ne sera plus précédée du mot clé abstract
- le mot clé final signifie que la méthode ne peut pas être polymorphée, i.e qu'en cas d'héritage on ne peut pas la redéfinir.

Méthode (2)

- le mot clé native est particulièrement très utile dès que l'on veut utiliser des fonctions écrites dans un autre langage que Java, comme par exemple utiliser des fonctions système du langage C. Dans le langage Java, on peut intégrer des fonctions écrites dans un autre langage tel que le langage C,... en utilisant ce mot clé
- le mot clé synchronized sert dans le cas où l'on veut utiliser des threads. Il permet à 2 ou plusieurs méthodes de synchroniser leurs accès aux informations (i.e. lorsqu'elle doivent accéder aux mêmes données).

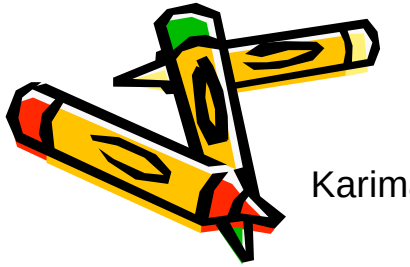


Types d'autorisation d'accès (1)

- lorsque l'on déclare une variable ou une méthode, il faut spécifier pour le champ access la visibilité de la variable ou de la méthode, qui peut être de type public, private ou protected
- public
  - ✓ le mot clé public signifie que la variable ou la méthode est *" visible"* de l'extérieur.
  - ✓ la variable peut alors être accédée et modifiée par n'importe quelle méthode se trouvant à l'extérieur de la classe, en faisant

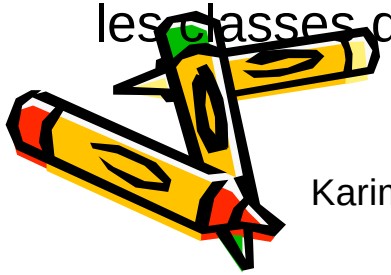
    nomObjet.laVariable
  - ✓ la méthode peut être appelée par n'importe quelle autre méthode se trouvant à l'extérieur de la classe, en faisant

    nomObjet.laMethode()

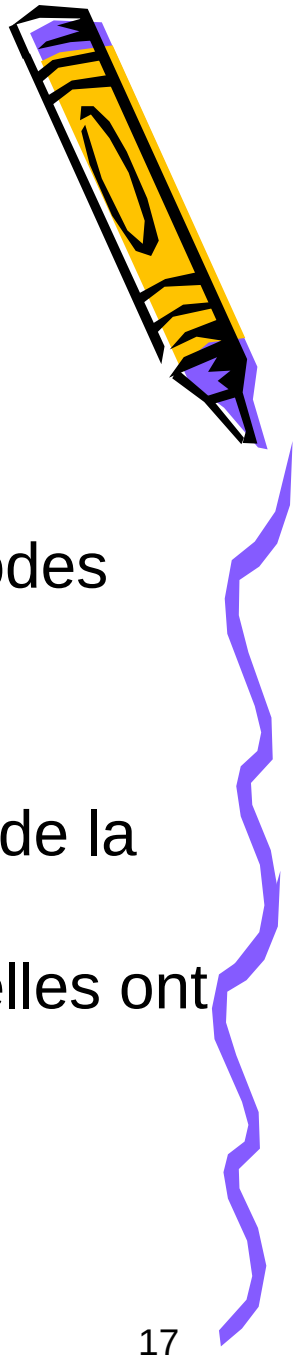


Types d'autorisation d'accès (2)

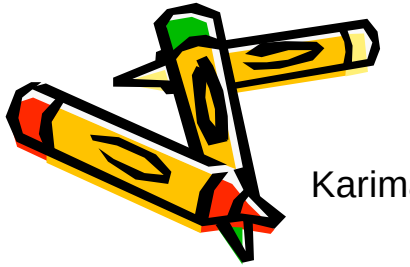
- private
  - ✓ signifie que la variable ne peut être modifiée qu'à l'intérieur de la classe
  - ✓ signifie que la méthode ne peut être appelée que par d'autres méthodes de la même classe
  - ✓ les sous-classes ne peuvent pas accéder aux variables et méthodes déclarées en private
- protected
  - ✓ signifie que la variable est visible par toutes les classes du package dont fait parti la classe où est défini la variable.
  - ✓ signifie que la méthode est visible par toutes les classes du package dont fait parti la classe où est défini cette classe.
  - ✓ les variables et les méthodes peuvent être également accéder par les sous-classes faisant partie d'autres packages.
- si rien n'est spécifié, cela signifie que les variables et les méthodes de la classe dans laquelle elles sont déclarées ne sont accessible que par les classes qui font parti du même *package*



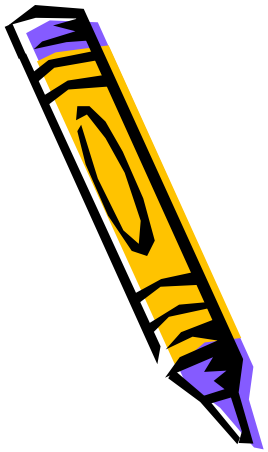
Variables



- **Variables d'instance**
  - sont déclarées à l'intérieur d'une classe
  - sont déclarées à l'extérieur de toute méthode
  - conservent l'état d'un objet, instance de la classe
  - sont accessibles et partagées par toutes les méthodes
- **Variables locales**
  - sont déclarées à l'intérieur d'une méthode
  - conservent une valeur utilisée pendant l'exécution de la méthode
  - ne sont accessibles que dans le bloc dans lequel elles ont été déclarées

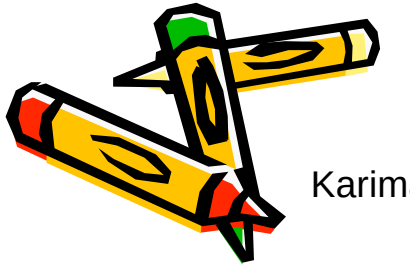


Constructeurs (1)



- un constructeur sert
  - ✓ à créer les instances
  - ✓ à initialiser les variables de ces instances (i.e leur état)
- un constructeur
  - ✓ a le même nom que la classe et n'a pas de type de retour
- une classe peut avoir un ou plusieurs constructeurs
  - ✓ si elle a plusieurs constructeurs, on parle de surcharge de constructeur
  - ✓ exemple:

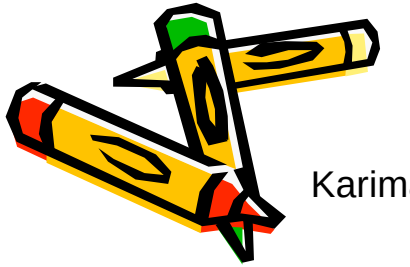
```
public class Etudiant {
    private String leNom, lePrenom;
    private int leNumEtudiant;
    public Etudiant (String unNom, String unPrenom) {
        leNom = unNom; lePrenom=unPrenom; leNumEtudiant=0;}
    public Etudiant (String unNom, String unPrenom,
                     int unNumEtudiant) {
        leNom = unNom; lePrenom = unPrenom;
        leNumEtudiant = unNumEtudiant;}
}
```



Constructeurs (2)

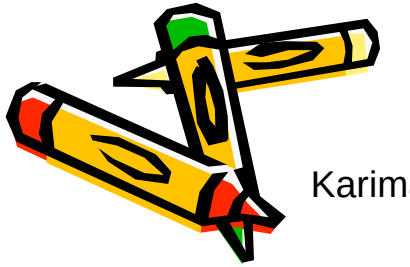
- une classe peut avoir un constructeur par défaut, i.e un constructeur sans argument
  - ✓ dans ce cas, les variables d'instance sont initialisées aux valeurs par défaut
  - ✓ exemple:

```
public class Etudiant {
    public static final String LE_NOM =  "inconnu";
    public static final String LE_PRENOM = "inconnu";
    private static int leNombreEtudiants = 0;
    private String leNom, lePrenom;
    private int leNumEtudiant;
    public Etudiant () {
        leNom = LE_NOM ; lePrenom = LE_PRENOM;
        leNumEtudiant = 0;
    }
}
```



Accesseurs et modificateurs

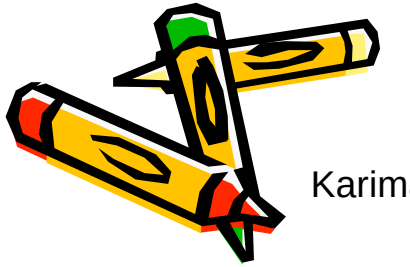
- les accesseurs et modificateurs sont des méthodes particulières qui servent à accéder à des données privées,
  i.e déclarées en private
- il existe deux types de méthodes:
  - ✓ les accesseurs en lecture, dit aussi *accesseur*
  - ✓ les accesseurs en modification, dit aussi *modificateur*
- pour les accesseurs en lecture, on utilise la notation suivante:
  - ✓ **toto.getLeNom()**
  - ✓ la méthode getLeNom() n'a pas de paramètres d'entrée
- pour les accesseurs en modification, on utilise la notation suivante :
  - ✓ **toto.setLeNumEtudiant(unNumEtudiant)**
  - ✓ la méthode setLeNumEtudiant(unNumEtudiant) a généralement un paramètre d'entrée qui servira à affecter la nouvelle valeur à la variable d'instance que l'on veut modifier



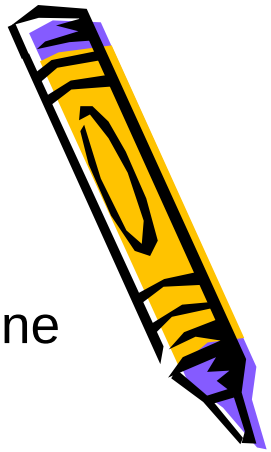
Méthodes (1)

- les accesseurs et modificateurs sont des méthodes particulières
- il existe d'autres méthodes qui permettent de faire d'autres traitements(i.e faire autre chose que lire et modifier une variable) par exemple faire le calcul d'une moyenne, le calcul d'une facture,…
- il existe des méthodes private qui servent de "sous-programmes" utilitaires aux autres méthodes de classe
- une méthode peut avoir zéro ou plusieurs paramètres d'entrée
- une méthode peut avoir zéro ou un seul paramètre de retour
- en ce qui concerne le nom de la méthode on essaye de lui donner un nom parlant. Par exemple pour calculer une moyenne de notes on l'appellerait calculerMoyenne()
- pour appeler ensuite la méthode, il faut faire référence à l'objet (i.e l'instance de classe à qui appartient la méthode) :

```
toto.calculerMoyenne();
```



Méthodes (2)



❍ Surcharge d'une méthode

- en Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a la même signature qu'une méthode existante

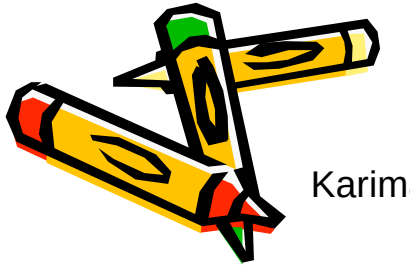
  public double calculerMoyenne()

  public double calculerMoyenne(double [] desNotes)

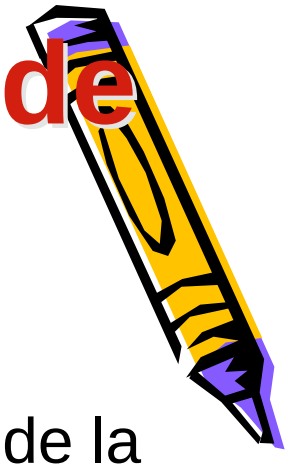
- en Java, il est interdit de surcharger une méthode en changeant le type de retour
- autrement dit, deux méthodes ne peuvent avoir la même signature sans avoir le même type de retour
- exemple : il est interdit d'avoir ces 2 méthodes dans une même classe :

  int calculerMoyenne()

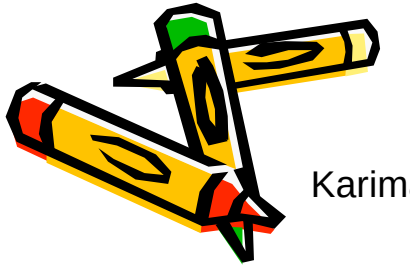
  double calculerMoyenne(double [] desNotes)



Utilisation des membres de classe



- **Création d'objets**
  - pour créer un objet, il faut utiliser l'opérateur new, de la manière suivante : Etudiant toto = new Etudiant()
- **Utilisation des membres de classes**
  - pour les variables : toto.leNom
  - pour les méthodes : toto.getLeNom()



Variables de classe

❍ Variables de classe

- le mot clé static signifie que la variable est une variable de classe
- les variables de classe sont partagées par toutes les instances d'une classe
- une variable de classe peut être initialisée lors de sa déclaration. Dans ce cas cette variable est initialisée une seule fois quand la classe est chargée en mémoire (i.e dès que l'on fait appel à la classe)
- l'utilisation des 2 mots clés public static, est réservé aux constantes

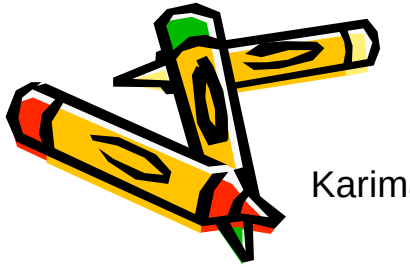
```
public static final String LE_NOM = "inconnu";
```

- une constante peut être utilisée depuis une autre classe. Pour cela il faut pécéder le nom de la variable par le nom de la classe

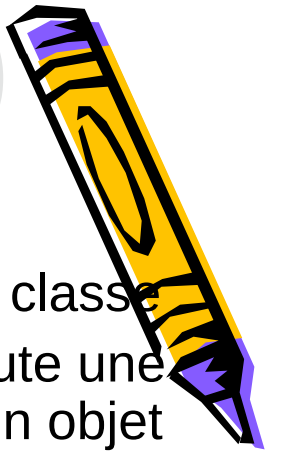
```
String leNom = Etudiant.LE_NOM
```

- une variable déclarée en private static est partagé par toutes les instances et n'est visible que dans sa propre classe

```
private static int leNombreEtudiants = 0
```



Méthodes de classe (1)



❍ Méthodes de classe

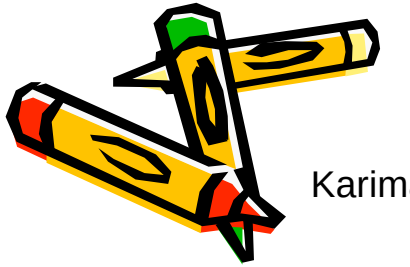
- le mot clé static signifie que la méthode est une méthode de classe
- une méthode ne peut être déclarée en static que si elle exécute une action qui est indépendante d'une instance de la classe (i.e un objet spécifique)
- exemple :

```
public static int nombreEtudiants() {
    return leNombreEtudiants;
}
```

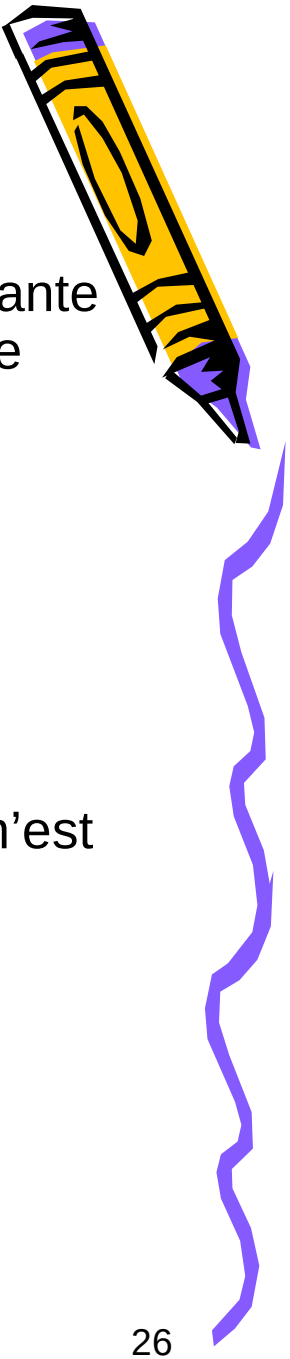
- pour appeler une méthode static depuis une autre classe, il faut précéder le nom de la méthode par le nom de la classe :

```
int leNombre = Etudiant.nombreEtudiants()
```

- une méthode de classe doit avoir une signature différente de celle d'une méthode d'instance



Méthodes de classe (2)



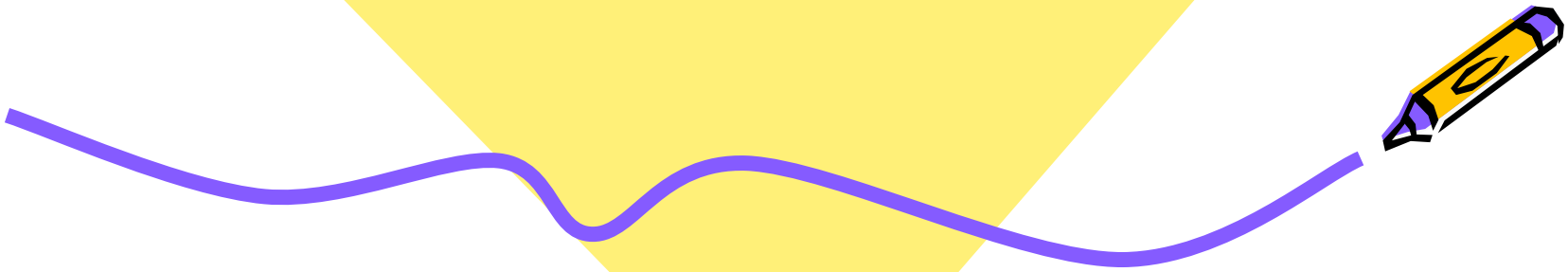
❍ Méthodes de classe

- comme une méthode de classe exécute une action indépendante d'une instance particulière de la classe, elle ne peut utiliser de référence à une instance courante (**this**)
- exemple, il serait interdire d'écrire

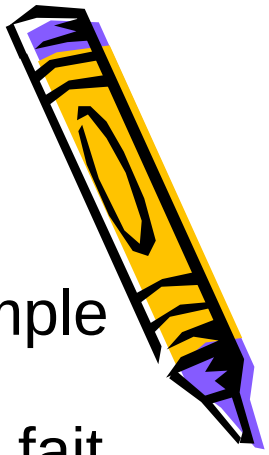
```
static double tripleSalaire() {
    return salary *3;
}
```

- la méthode main () est nécessairement static
- elle est exécutée au début du programme. Aucune instance n'est donc déjà créée lorsque la méthode main () commence son exécution. Ça ne peut donc pas être une méthode d'instance

Héritage



Héritage (1)



Définition

- on utilise l'héritage lorsqu'on définit un objet, par exemple Etudiant qui «est un» autre objet de type par exemple Personne avec plus de fonctionnalités qui sont liés au fait que l'objet soit un étudiant
- l'héritage permet de réutiliser dans la classe Etudiant le code de la classe Personne (lorsque l'on dit que Etudiant hérite de Personne) sans toucher au code initial : on a seulement besoin du code compilé
- l'héritage minimise les modifications à effectuer : on indique seulement ce qui a changé dans Etudiant par rapport au code de Personne, on peut par exemple
  - ✓ rajouter de nouvelles variables
  - ✓ rajouter de nouvelles méthodes
  - ✓ modifier certaines méthodes

Héritage (2)

Vocabulaire

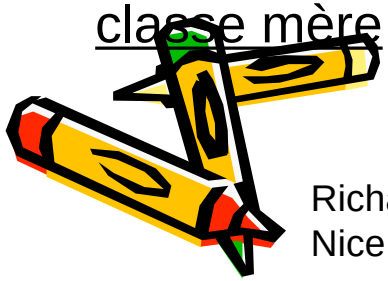
- la classe Personne s'appelle une classe mère, classe parente ou super-classe
- la classe Etudiant qui hérite de la classe Personne s'appelle une classe fille ou sous-classe
- pour désigner une classe mère, on utilise le mot clé extends

```
class Etudiant extends Personne
```

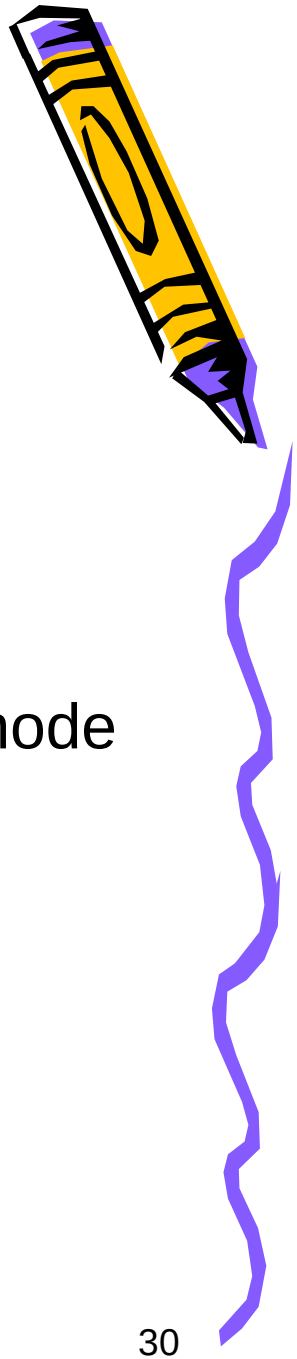
- par défaut, une classe (qui n' a pas d'extends dans sa définition) hérite de la classe Object, qui est la superclasse de toutes les classes
- exemples d'héritage

```
class Voiture extends Vehicule
class Employe extends Personne
```

- le mot clé super désigne la superclasse
- Une classe ne peut hériter que d'une seule classe et n'a donc qu'une seule classe mère



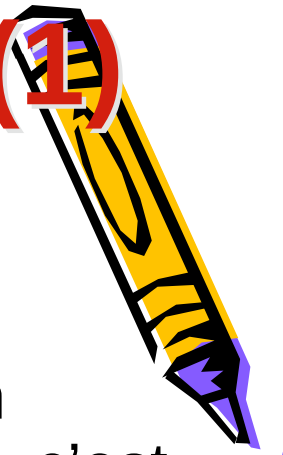
Héritage (3)



❍ Ce que peut faire une classe fille

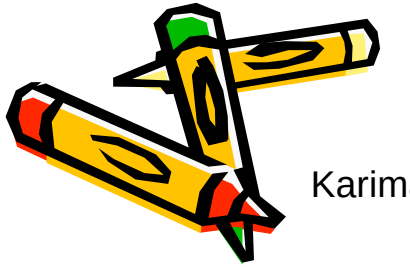
- la classe qui hérite peut
  - ajouter des variables,
  - ajouter des méthodes et des constructeurs
  - redéfinir des méthodes
  - surcharger des méthodes
- mais elle ne peut pas retirer une variable ou une méthode

Surcharge et redéfinition (1)

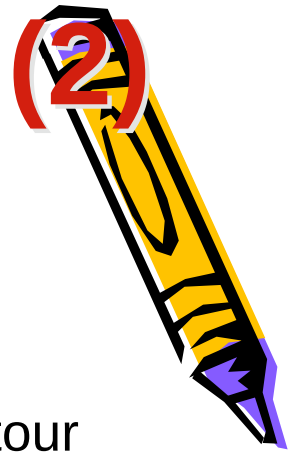


Surcharger une méthode

- une classe fille peut *surcharger* une méthode
- ce qui signifie qu'elle peut réécrire une méthode en gardant le même nom mais pas la même signature, c'est-à dire avec des paramètres d'entrée différents mais même paramètre de retour (possible aussi à l'intérieur d'une classe, voir précédemment)
- lorsque l'on surcharge une méthode,
  - cela signifie que l'on ajoute un comportement
  - et que l'on modifie son interface



Surcharge et redéfinition (2)



Redéfinir une méthode

- une classe fille peut *redéfinir* une méthode
- ce qui signifie qu'elle peut réécrire une méthode en gardant exactement la même signature et le même paramètre de retour
- lorsque l'on redéfinit une méthode
  - cela signifie que l'on modifie son implémentation
  - mais que l'on ne change pas son interface
- pour appeler une méthode de la super-classe, on doit précéder le nom de la méthode par le mot clé super

```
super.getNom()
```

- on ne peut pas appeler une méthode de la super-super-classe, i.e on ne peut pas faire `super.super.getNom()`
- lorsque l'on redéfinit une méthode on ne peut pas réduire le niveau d'accès de la méthode, c'est-à-dire on ne peut pas par exemple redéfinir une classe public en une classe private

This et Super

This

- le mot clé this désigne l'objet courant;

  this.leNombre = *leNombre* //la variable leNombre précédée de this est la variable d'instance de cet objet, alors que *leNombre* sans le this représente une variable quelconque qui n'est pas la variable d'instance de l'objet pointé par this

  this.getPrix() //permet d'appeler la méthode getPrix() définie dans cette classe

  return this //retourne l'objet courant

Super

- le mot clé super désigne la superclasse;

super.getPrix() // permet d'appeler la méthode getPrix() définie dans la superclasse

Cas particulier des constructeurs

- un constructeur peut appeler un autre constructeur
  - this(arg1,…) //appel du constructeur d'argument arg1,…de l'objet en cours de construction. Il doit être en 1ère ligne
  - super(arg1,…) //appel du constructeur d'argument arg1,…de la superclasse. Il doit être en 1ère ligne du constructeur