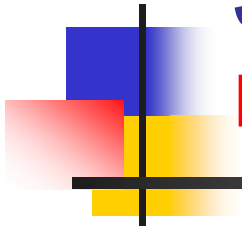
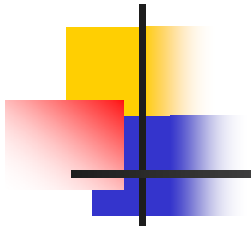


Java

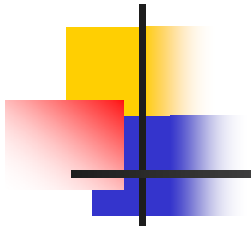
Licence professionnelle CISI 2009-2010

Cours 10 : Type générique



Introduction

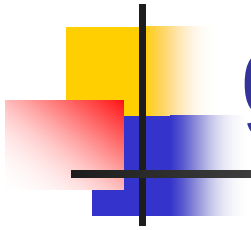
- *La programmation générique*
  - *nouveauté la plus significative du langage de programmation Java depuis la version 1.0.*
  - *permet d'écrire du code plus sûr et plus facile à lire qu'un code parsemé de variables Object et de cast*
  - *particulièrement utile pour les classes de collection, comme ArrayList*



Introduction

- Pourquoi la programmation générique ?
  - *La programmation générique implique d'écrire du code qui puisse être réutilisé pour des objets de types différents*
  - *Nous n'avons plus besoin, par exemple, de programmer des classes différentes pour collecter les objets String et File, puisque dorénavant, la classe ArrayList collecte les objets de n'importe quelle classe*

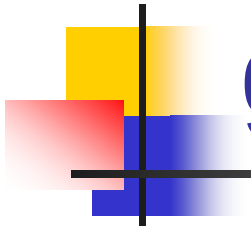
Pourquoi la programmation générique ?



- Avant le JDK 5.0
  - La programmation équivalente à la programmation générique s'obtenait à chaque fois par le mécanisme d'héritage
  - Ainsi, la classe ArrayList conservait simplement un tableau de références Object (la classe Object est l'ancêtre de toutes les autres classes) :

```
public class ArrayList { // Avant le JDK 5.0
   public Object get(int indice) { ... }
   public void add(Object élément) { ... }
            ...
   private Object[] tableauElémentsStockés;
}
```

Pourquoi la programmation générique ?



- Avant le JDK 5.0
  - Cette approche présente deux problèmes
  - D'une part, il faut avoir recours au transtypage (cast) lorsque vous récupérez une valeur :

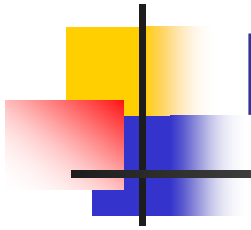
```
ArrayList fichier = new ArrayList() ;
...
String nomFichier = (String) fichier.get(0) ;
```

Pourquoi la programmation générique ?

- Avant le JDK 5.0
  - D'autre part, il n'existe aucune procédure de vérification des erreurs
  - Vous pouvez ajouter des valeurs de n'importe quelle classe :

    ```
    fichier.add(new File("...")) ;
    ```

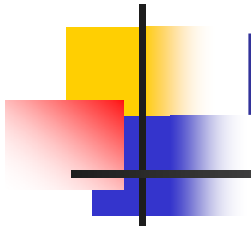
  - Cet appel compile et s'exécute sans erreur
  - Par ailleurs, transtyper le résultat de get() sur une chaîne produira une erreur, puisque normalement, nous devrions récolter un objet de type File



La programmation générique

- A partir du JDK 5.0
  - Le JDK 5.0 propose une meilleure solution
    - les paramètres de type
  - La classe ArrayList est désormais paramétrée et dispose d'un paramètre qui doit être un type d'objet
  - Ainsi, à chaque fois qu'on utilise la classe ArrayList
    - on doit systématiquement préciser le type d'élément qui doit être stocké à l'intérieur de cette collection :

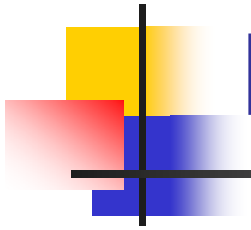
```
ArrayList<String> fichier = new ArrayList<String>() ;
```



La programmation générique

- *Votre code est désormais plus facile à lire*
- *Nous voyons immédiatement que cet ArrayList sera composée uniquement d'objets de type String*
- *Le compilateur utilisera également ces informations à bon escient*
- *Aucun transtypage n'est nécessaire pour appeler la méthode get() :*
  - *grâce à notre déclaration, le compilateur sait que le type de retour est String, et non Object :*

```
String nomFichier = fichier.get(0) ;
```



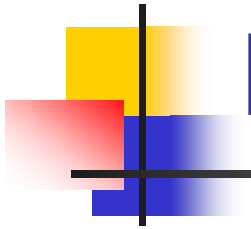
La programmation générique

- Le compilateur sait également que la méthode d'un ArrayList<String> possède un paramètre de type String
  - Cette technique est bien plus sûre que d'utiliser systématiquement un paramètre de type Object
- Ainsi, le compilateur peut vérifier que vous n'insérez aucun objet d'un type érroné
- Par exemple, l'instruction suivante :
  - fichier.add(new File("...")) ; // ne peut ajouter que des objets
    //String à un ArrayList<String>
- ne se compilera pas
  - *Une erreur de compilation vaut bien mieux qu'une exception de transtypage de classe au moment de l'exécution*

La programmation générique

- Définition d'une classe générique simple
  - *Une classe générique est une classe comprenant une ou plusieurs variables de type = une classe paramétrée, dont le paramètre est un type d'objet*
  - *Exemple*

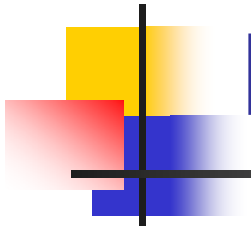
```
public class Paire<T> {
    private T premier;
    private T deuxième;
    public Paire() { premier=null; deuxième=null;}
    public Paire(T premier, T deuxième) { this.premier=premier;
            this.deuxième=deuxième; }
    public T getPremier() {return this.premier; }
    public void setPremier(T premier) { this.premier = premier; }
    public T getDeuxième() {return this.deuxième; }
    public void setDeuxième(T deuxième) {this.deuxième = deuxième; }
}
```



La programmation générique

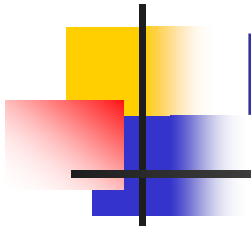
Définition d'une classe générique simple

- La classe Paire introduit une variable de type T, incluse entre <> après le nom de la classe
- Une classe générique (paramétrée) peut posséder plusieurs variables de type
  - utiliser l'opérateur virgule pour les séparer
- *Les variables de type peuvent être utilisées tout au long de la définition de la classe pour spécifier le type de retour des méthodes, le type des attributs, ou même le type de certaines variables locales*
- *rien n'empêche également d'utiliser des attributs ou d'autres éléments avec des types bien définis, c'est-à-dire non paramétrable, comme int, String, etc.*



La programmation générique

- Utilisation de la classe générique
  - *L'attribut premier utilise une variable de type*
  - *Ainsi, comme son nom l'indique premier pourra être de n'importe quel type*
  - *C'est celui qui utilisera cette classe qui spécifiera le type qu'il désire utiliser pour décrire l'objet de cette classe Paire*
  - Ainsi, vous instanciez une classe paramétrée en précisant le type que vous désirez prendre



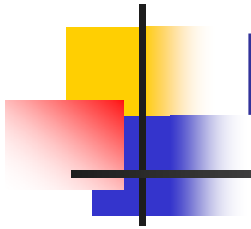
La programmation générique

Utilisation de la classe générique

- Par exemple :

  ```
  Paire<String> ordre ;
  ```

- Ici, ordre est un objet de type Paire<String>
- Le paramètre de la classe Paire, ou exprimé autrement, la variable de type est String
- Ainsi, au moment de cette déclaration, à l'intérieur de la classe Paire, tout ce qui fait référence à T est remplacé par le véritable type, c'est-à-dire String

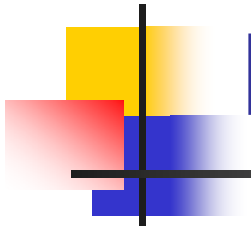


La programmation générique

Utilisation de la classe générique

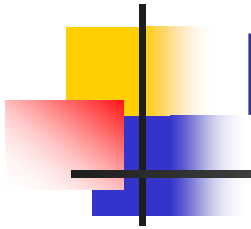
- C'est comme si nous avions, pour cette utilisation, la classe suivante :

```
public class Paire {
    private String premier;
    private String deuxième;
    public Paire() { premier=null; deuxième=null;}
    public Paire(String premier, String deuxième) { this.premier=premier;
            this.deuxième=deuxième; }
    public String getPremier() {return this.premier; }
    public void setPremier(String premier) { this.premier = premier; }
    public String getDeuxième() {return this.deuxième; }
    public void setDeuxième(String deuxième) {this.deuxième = deuxième;
    }
}
```



La programmation générique

- Exemple d'utilisation : Test.java
  - *Le programme suivant met en oeuvre la classe Paire*
  - *La méthode minmax() statique parcourt un tableau de chaînes de caractères et calcule en même temps la valeur minimale et la valeur maximale*
  - *Elle utilise un objet Paire pour renvoyer les deux résultats*



La programmation générique

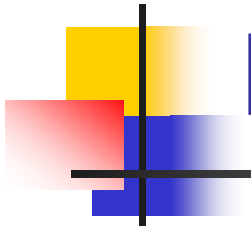
- Exemple d'utilisation : Test.java
  - *Le programme suivant met en oeuvre la classe Paire*
  - *La méthode minmax() statique parcourt un tableau de chaînes de caractères et calcule en même temps la valeur minimale et la valeur maximale*
  - *Elle utilise un objet Paire pour renvoyer les deux résultats*

```
public class Test {
    public static void main(String[] args) {
            String[] phrase = {"Marie", "possède", "une", "petite", "lampe"};
            Paire<String> extrêmes = TableauAlg.minmax(phrase);
            System.out.println("min = "+extrêmes.getPremier());
            System.out.println("max = "+extrêmes.getDeuxième());
    }
}
```

La programmation générique

```
class TableauAlg {
    public static Paire<String>
    minmax(String[] chaînes) {
      if (chaînes==null ||
      chaînes.length==0) return null;
      String min = chaînes[0];
      String max = chaînes[0];
      for (String chaîne : chaînes) {
            if (min.compareTo(chaîne) >
            0) min = chaîne;
            if (max.compareTo(chaîne) <
            0) max = chaîne;}
            return new Paire<String>(min,
            max);
    }
}
```

```
class Paire<T> {
    private T premier;
    private T deuxième;
    public Paire() { premier=null;
    deuxième=null;}
    public Paire(T premier, T deuxième)
          { this.premier=premier;
          this.deuxième=deuxième; }
    public T getPremier() {return
          this.premier; }
    public void setPremier(T premier) {
    this.premier = premier; }
    public T getDeuxième() {return
    this.deuxième; }
    public void setDeuxième(T
    deuxième) {this.deuxième =
    deuxième; }
}
```



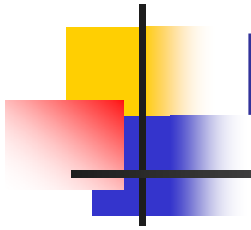
La programmation générique

Méthodes génériques

- On vient de voir comment définir une classe générique
- Il est aussi possible de définir une seule méthode avec des paramètres de type :

```
class TableauAlg {
    public static <T> T getMilieu(T[] tableau) {
            return tableau[tableau.length / 2];
    }
}
```

- Cette méthode est définie dans une classe ordinaire, et non dans une classe générique
- C'est toutefois une méthode générique
- Les variables de type sont insérées après les modificateurs (public static, dans ce cas) et avant le type de retour



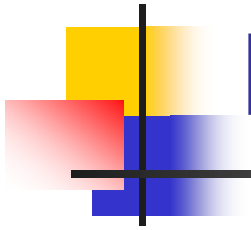
La programmation générique

Méthodes génériques

- **Lorsque on appelle une méthode générique, on peut placer les types réels, entourés des signes <>, avant le nom de la méthode :**

```
String[] noms= {"Marie", "possède", "une", "petite", "lampe"};
String milieu = TableauAlg.<String>getMilieu(noms) ;
```

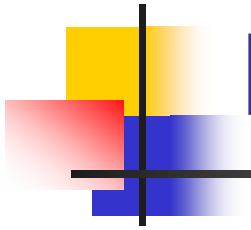
- ***Dans ce cas, on peut omettre le paramètre type <String> de l'appel de méthode***
- ***Le compilateur dispose de suffisamment d'informations pour un déduire la méthode que l'on souhaite utiliser***
- ***Il fait correspondre le type des noms (donc, String[]) avec le type générique T[] et en déduit que T doit être un String)***
- ***On peut donc simplement appeler de la manière suivante :***
  - **String milieu = TableauAlg.getMilieu(noms) ;**



La programmation générique

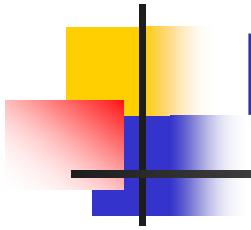
- Limites pour les variables de type
  - *Par moment, une classe paramétrée ou une méthode paramétrée doit placer des restrictions sur des variables de type.*
  - *Visualisons le problème au travers d'un exemple :*

```
class TableauAlg {
    public static <T> T min(T[] tab) {
            if (tab==null || tab.length==0) return null;
            T pluspetit = tab[0];
            for (T val : tab)
            if (pluspetit.compareTo(val) > 0) pluspetit = val;
            return pluspetit;
    }
}
```



La programmation générique

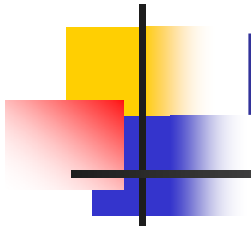
- Limites pour les variables de type (suite)
  - **Mais dans cette méthode générique min() un problème demeure**
  - **En effet, la variable plus petite possède un type T, ce qui signifie qu'il pourrait s'agir d'un objet d'une classe ordinaire**
  - **Comment savoir alors que la classe à laquelle appartient T possède une méthode compareTo() ?**
  - ***D'ailleurs, si on écrit ce code, on obtient une erreur du compilateur en spécifiant que cette méthode compareTo() n'est pas connue pour un type quelconque T***



La programmation générique

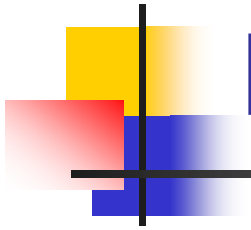
- Limites pour les variables de type (suite)
  - **La solution consiste à restreindre T à une classe qui implémente l'interface Comparable**
  - **Pour y parvenir, on doit donner une limite pour la variable de type T :**

```
class TableauAlg {
        public static <T extends Comparable> T min(T[] tab) {
        ...
        }
}
```



La programmation générique

- Limites pour les variables de type (suite)
  - **Désormais, la méthode générique min() ne peut être appelée qu'avec des tableaux de classes qui implémentent l'interface Comparable, comme String, Date, etc.**
    - **Appeler min() avec un tabelau de Rectangle produit une erreur de compilation car la classe Rectangle n'implémente pas Comparable**
    - ***Nous pouvons nous demander pourquoi utiliser le mot clé extends plutôt que le mot clé implements dans cette situation (après tout, Comparable est une interface)***

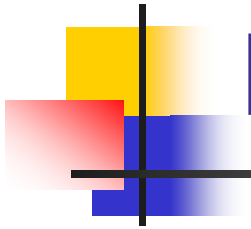


La programmation générique

- La notation :

  <T extends TypeLimitant>

- indique que T doit être un sous-type du type limitant
- *T et le type limitant peuvent être une classe ou une interface*
- *Le mot clé extends a été choisi car il constitue une approximation raisonnable du concept de sous-type et que les concepteurs Java ne souhaitaient pas ajouter un nouveau mot clé (comme sub)*

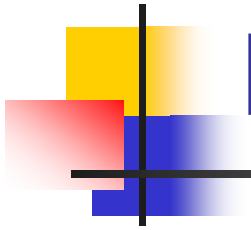


La programmation générique

- Une variable peut avoir plusieurs limites :

  <T extends Comparable & Serializable >

- Les types limitants sont séparés par des esperluettes (&) car les virgules sont utilisées pour séparer les variables de type
- *Comme pour l'héritage Java, on peut disposer d'autant d'interfaces qu'on le souhaite, mais une seule des limites peut être une classe*
- *De plus, si on dispose d'une classe agissant comme élément limitant, elle doit figurer en première place dans la liste*



La programmation générique

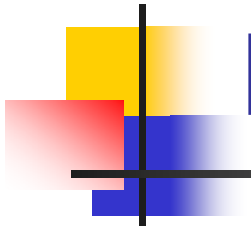
- Exemple d'utilisation
  - *Nous reprenons l'exemple précédent en réécrivant la méthode minmax() de manière à la rendre générique*
  - *La méthode calcule le minimum et le maximum d'un tableau générique et renvoie un Paire<T>*

```
public class Test {
    public static void main(String[] args) {
            String[] phrase = {"Marie", "possède", "une", "petite",
            "lampe"};
            Paire<String> extrêmes = TableauAlg.minmax(phrase);
            System.out.println("min = "+extrêmes.getPremier());
            System.out.println("max = "+extrêmes.getDeuxième());
    }
}
```

La programmation générique

```
class TableauAlg {
    public static <T extends Comparable>
    Paire<T> minmax(T[] tab) {
        if (tab==null || tab.length==0)
        return null;
        T min = tab[0];
        T max = tab[0];
        for (T élément : tab) {
        if (min.compareTo(élément)    >
    0) min = élément;
        if (max.compareTo(élément)   <
    0) max = élément;
        }
        return new Paire<T>(min, max);
    }
}
```

```
class Paire<T> {
    private T premier;
    private T deuxième;
    public Paire() { premier=null;
    deuxième=null;}
    public Paire(T premier, T deuxième)
    { this.premier=premier;
    this.deuxième=deuxième; }
    public T getPremier() {return
    this.premier; }
    public void setPremier(T premier) {
    this.premier = premier; }
    public T getDeuxième() {return
    this.deuxième; }
    public void setDeuxième(T
    deuxième) {this.deuxième =
    deuxième; }
}
```

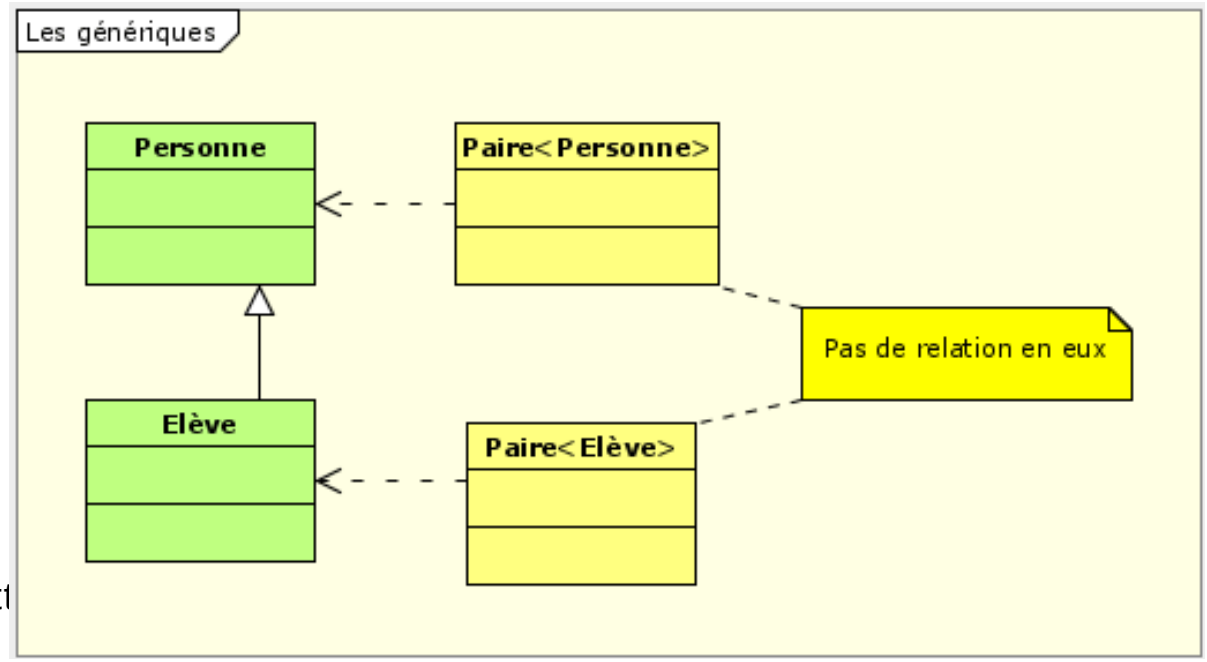


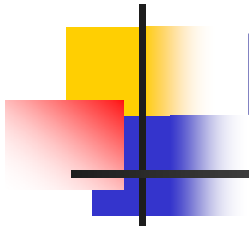
La programmation générique

- *Règles d'héritage pour les types génériques*
  - *Soit une classe de base (ancêtre) Personne et une sous-classe (classe dérivée) Elève qui hérite donc de Personne*
  - *La question qui se pose "Est-ce que Paire<Elève> est une sous-classe de Paire<Personne> ?".*
  - Bizarrement, la réponse est "Non«
  - Par exemple, le code suivant ne sera pas compilé :
    - Elève[] élèves = ... ;
    - Paire<Personne> personne = TableauAlg.minmax(élèves) ;

La programmation générique

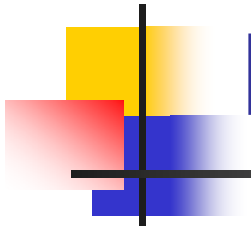
- *Règles d'héritage pour les types génériques*
  - La méthode minmax() renvoie un Paire<Elève>, et non pas un Paire<Personne>, et il n'est pas possible d'affecter l'une à l'autre
  - En général, il n'existe pas de relation entre Paire<S> et Paire<T>, quels que soient les éléments auxquels S et T sont reliés





La programmation générique

- Types joker
  - *Comme nous venons juste de le découvrir au niveau de l'héritage, un système de type trop rigide n'est pas toujours très agréable à utiliser. Les concepteurs Java ont*
  - *inventé une "sortie de secours" ingénieuse (tout en étant sûre) : le type joker.*
  - *Par exemple, le type joker :*
    - **<? extends Personne>**
  - remplace toute paire générique dont le paramètre type est une sous-classe de Personne, comme Paire<Elève>, mais pas, bien entendu, Paire<String>



La programmation générique

- Types joker
  - **Imaginons que nous voulions écrire une méthode qui affiche des paires de personnes, comme ceci :**

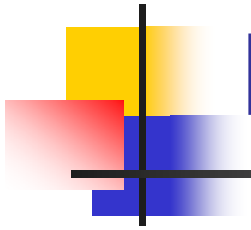
```
public static void afficheBinômes(Paire<Personne> personnes) {
    Personne premier = personnes.getPremier();
    Personne deuxième = personnes.getDeuxième();
    System.out.println(premier.getNom()+" et
    "+deuxième.getNom()+" sont ensembles.");
}
```

avec :

La programmation générique

```
class Paire<T> {
    private T premier;
    private T deuxième;
    public Paire() { premier=null;
    deuxième=null;}
    public Paire(T premier, T deuxième) {
            this.premier=premier;
            this.deuxième=deuxième; }
    public T getPremier() {return
    this.premier; }
    public void setPremier(T premier) {
    this.premier = premier; }
    public T getDeuxième() {return
    this.deuxième; }
    public void setDeuxième(T deuxième)
    {this.deuxième = deuxième; }
}
```

```
class Personne {
    private String nom, prénom;
    public String getNom() { return nom; }
    public String getPrénom() { return
    prénom; }
    public Personne(String nom, String
    prénom) { this.nom=nom;
    this.prénom=prénom;}
}
class Elève extends Personne {
    private double[] notes = new
    double[10];
    private int nombreNote = 0;
    public Elève(String nom, String
    prénom) { super(nom, prénom); }
    public void ajoutNote(double note) { if
    (nombreNote<10)
    notes[nombreNote++] = note; }
}
```

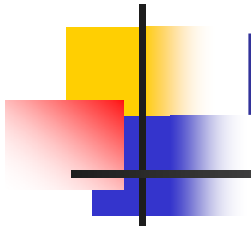


La programmation générique

- Comme nous l'avons vu dans le chapitre précédent, nous ne pouvons pas passer un Paire<Elève> à cette méthode afficheBinôme()
  - Voilà qui est plutôt limitatif
- La solution est simple pour résoudre ce problème, il suffit d'utiliser un joker :

```
public static void afficheBinômes(Paire<? extends Personne>
    personnes) {
    Personne premier = personnes.getPremier();
    Personne deuxième = personnes.getDeuxième();
    System.out.println(premier.getNom()+" et "+deuxième.getNom()+"
    sont ensembles.");
}
```

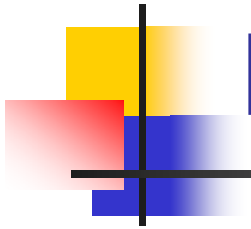
- ***Ici, nous indiquons que nous pouvons utiliser n'importe quelle classe qui fait partie de l'héritage de Personne, classe de base comprise***



La programmation générique

- **Voici, donc une utilisation possible :**

```
public class Main {
    public static void main(String[] args) {
        Personne personne1 = new Personne("Lagafe",
        "Gaston");
        Personne personne2 = new Personne("Talon", "Achile");
        Paire<Personne> bynômePersonne = new
        Paire<Personne>(personne1, personne2);
        Elève élève1 = new Elève("Guillemet", "Virgule");
        Elève élève2 = new Elève("Mouse", "Mickey");
        Paire<Elève> bynômeElève = new Paire<Elève>(élève1,
        élève2);
        afficheBinômes(bynômePersonne);
        afficheBinômes(bynômeElève);
    }
```



La programmation générique

```
public static void afficheBinômes(Paire<? extends
   Personne> personnes) {
        Personne premier = personnes.getPremier();
        Personne deuxième = personnes.getDeuxième();
        System.out.println(premier.getNom()+" et
        "+deuxième.getNom()+" sont ensembles.");
   }
}
```