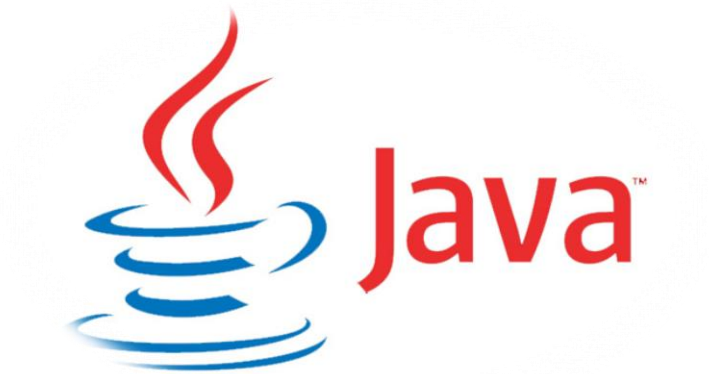
Heritage



Prof: Mme Sara SAIB

Définition

L'avantage essentiel d'un **langage orienté-objet** est que le code est **réutilisable**. Grâce à **l'héritage**, on peut faire dériver une nouvelle classe d'une classe existante et ainsi en récupérer les attributs et méthodes, sans avoir à la réécrire complètement.

Terminologie

• On dit que la classe *Enfant* **hérite** de la classe *Parent*, qu'elle **étend** cette ancienne classe.

• *Enfant* est alors **une sous-classe** (subclassed) de *Parent*.

• *Parent* est une **super-classe** (surclasse) de *Enfant*.

Définition

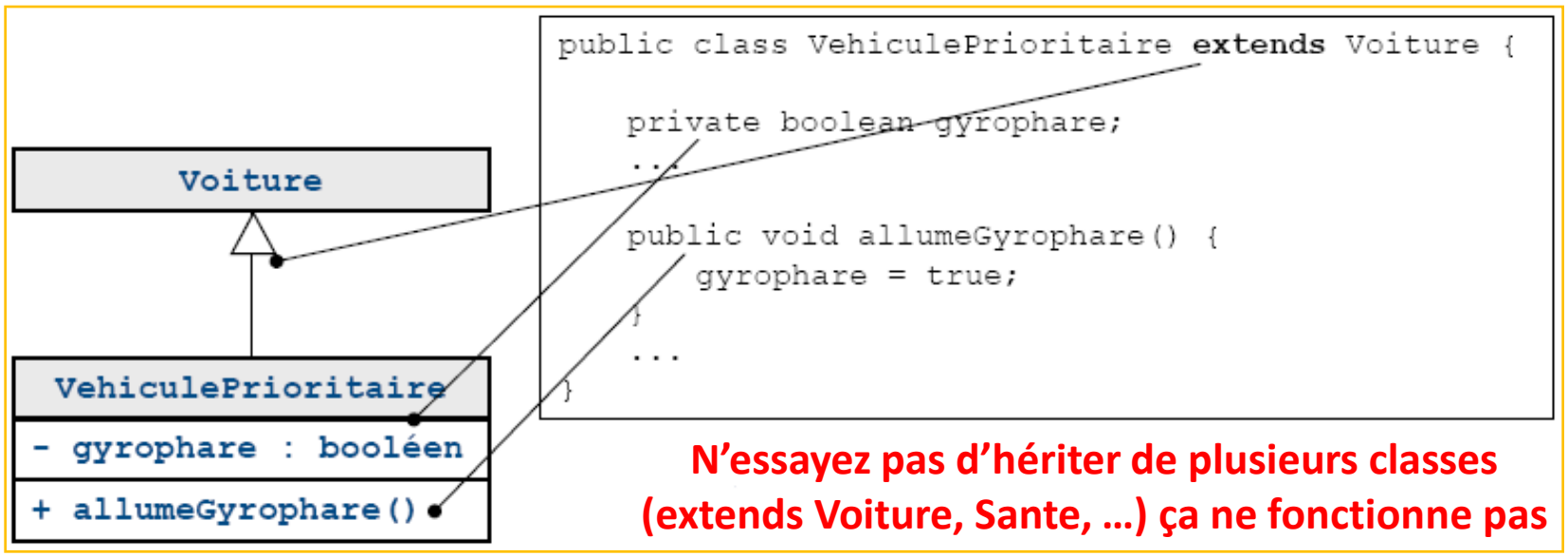
Une sous-classe **reprend toutes les propriétés et méthodes** de la super-classe, tout en pouvant intégrer de champs nouveaux (attributs et méthodes)

- **Remarque :** En Java, toutes les classes sont dérivées de la classe spéciale **Object**.
- C'est la racine de la hiérarchie des classes (cela entraine que toute classe Java possède déjà à sa naissance un certain nombre de variables et de méthodes).
- Dans la déclaration d'une classe si la clause **extends** n'est pas présente, la surclasse est donc **Object** directement.

L'héritage en java

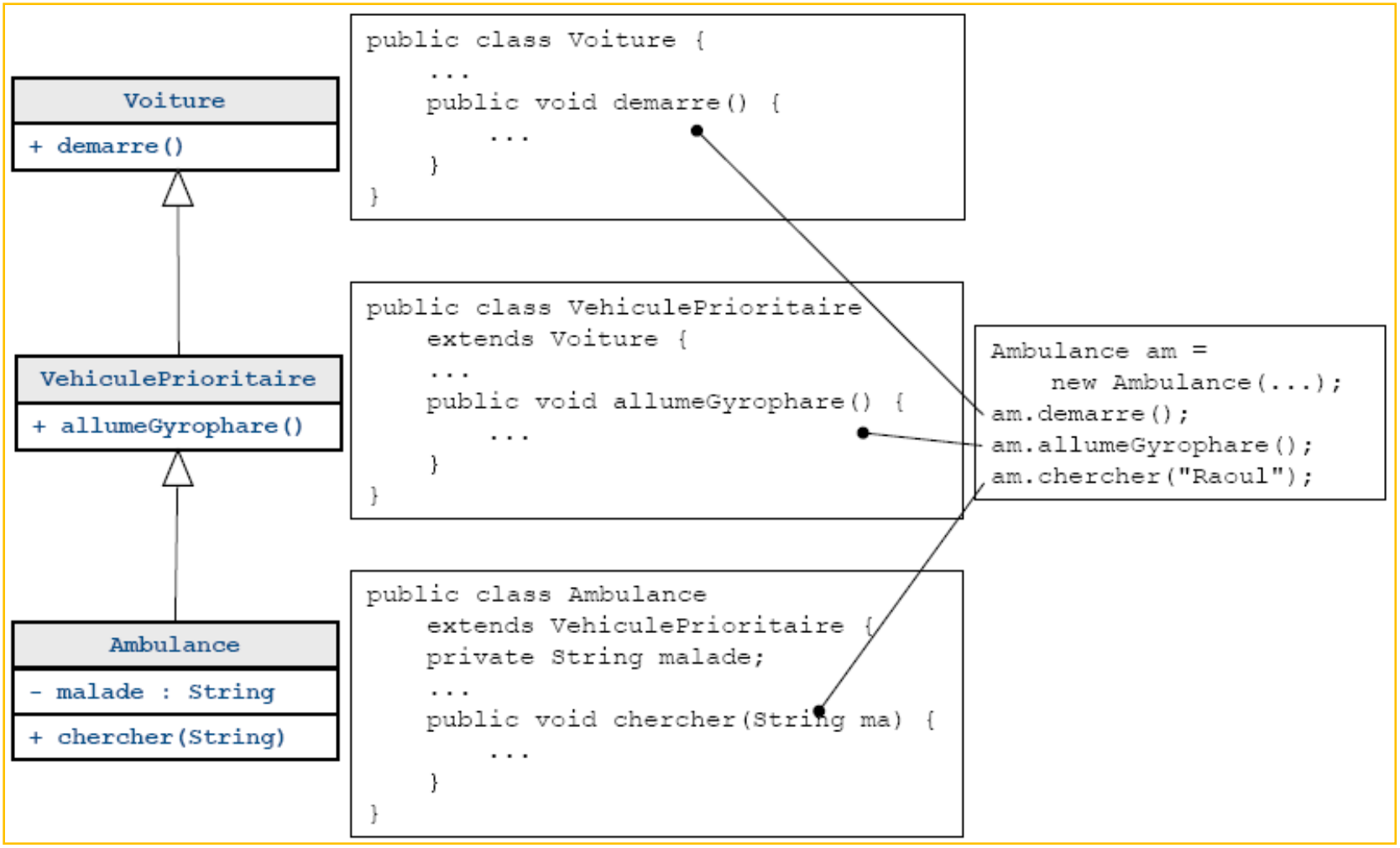
Héritage simple

- Une classe ne peut hériter que d'une seule autre classe
- Dans certains autres langages (ex : C++) possibilité d'héritage multiple
- Utilisation du mot-clé **extends après le nom de la classe**



N'essayez pas d'hériter de plusieurs classes (extends Voiture, Sante, …) ça ne fonctionne pas

L'héritage à plusieurs niveaux



La redéfinition des constructeurs et des méthodes

Lorsqu'on code la classe Enfant, on a la possibilité de remplacer une méthode de la super-classe Parent, en définissant une méthode avec le même nom et les mêmes paramètres. On appelle cette fonctionnalité **redéfinition des méthodes héritées**

La redéfinition de méthodes: **fonctionnalité** qui **autorise la ré-écriture d'une méthode héritée, en gardant le même nom** et **les mêmes paramètres.**

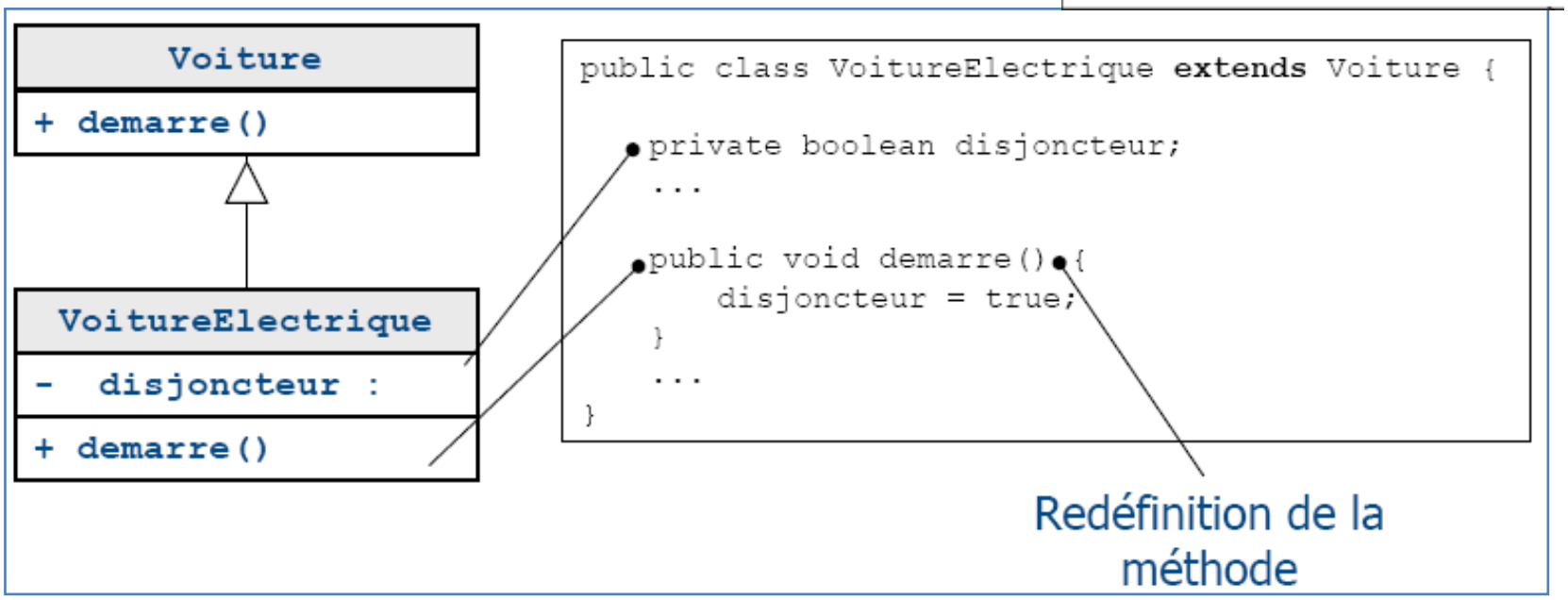
L'opération de redéfinition d'une méthode masque inévitablement la méthode d'origine de la superclasse Parent.

Ne pas confondre **la redéfinition de méthodes** et **la surcharge de méthodes**

La redéfinition des constructeurs et des méthodes

- Une voiture électrique est une voiture dont l'opération de démarrage est différente
- On démarre une voiture électrique en activant un disjoncteur

```
public class Voiture {
    ...
    public void demarre() {
        ...
    }
}
```



La redéfinition des constructeurs et des méthodes

Remarque

il n'est pas possible de **redéfinir un constructeur,** puisque celui-ci porte **le nom de la classe dans laquelle il est déclaré**, et que les noms des deux classes diffèrent obligatoirement.

La redéfinition avec réutilisation: mot clé super

- L’opération de redéfinition d'une méthode masque inévitablement la méthode d'origine de la superclasse Parent.

- Pour pouvoir faire appel tout de même à la méthode Parent (réutiliser le code de la méthode héritée) , on écrit le mot-clé **super** devant le nom de la méthode.

- Pour réutiliser le code de la méthode héritée on utilise le mot clé super. **Le mot clé super permet la désignation explicite de l’instance d’une** classe dont le type est celui de la classe mère

- Supposons qu'on ait redéfini dans la classe Enfant, la méthode **M()** déjà définie dans la classe Parent. Alors, **super.M(),** fait appel à la méthode **M()** définie dans la classe Parent et non à celle d'Enfant.

Usage des constructeurs

Comme les méthodes, il est possible de réutiliser le code des constructeurs de la super-classe.

On peut appeler explicitement un constructeur de la classe mère à l'intérieur d'un constructeur de la classe fille: Utiliser le mot-clé **super**

L'appel au constructeur de la superclasse doit se faire absolument en première instruction

super(paramètres du constructeur);

Appel implicite d'un constructeur de la classe mère est effectué quand il n'existe pas d'appel explicite. Java insère implicitement l'appel **super()**

Usage des constructeurs

```
public class Document {
    protected int num;
    protected String titre;

    public Document(int num, String titre){
        this.num=num;
        this.titre=titre;
    }
```

```
public class Livre extends Document {
    protected String auteur;

    public Livre(int num,String titre, String auteur ){
        super(num,titre);
        this.auteur=auteur;
        }
```

Appel d'un constructeur de la super classe

Usage des constructeurs

Rappel : si une classe ne définit pas explicitement de constructeur, elle possède alors un constructeur par défaut

- Sans paramètre
- Qui ne fait rien
- Inutile si un autre constructeur est défini explicitement

Usage des constructeurs

```java
public class Personne {
    protected int id;
    protected String nom;
    protected String prenom;

    private static int count;

    public Personne(String nom, String prenom) {
        this.id = ++count;
        this.nom = nom;
        this.prenom = prenom;
    }
```

Constructeur explicite: désactivation du constructeur par défaut

```java
public class Etudiant extends Personne {

    private String cne;

    public Etudiant(String nom, String prenom, String cne) {
        super();
        this.cne = cne;
    }
```

Erreur : il n'existe pas dans Personne de constructeur sans paramètre

Les données privées et l’héritage

Si un attribut de la classe Parent est déclarée private, il n'est visible que de sa propre classe, **même la classe Enfant n'y accède pas !**

```
public class Ville {
    private String nom;
    private int nbHabitants; .....

public class Capitale extends Ville
{ .....
```

Les données privées et l’héritage

Il peut être souhaitable d'avoir un accès direct à certaines variables d'instance de Parent, en lecture comme en modification, tout en les "protégeant" de l'extérieur, des autres classes. Il faut pour cela les définir avec la qualification **protected**

Les attributs ou les méthodes protected sont accessibles dans tout le package et dans les sous classes

Classe Abstraite

Les classes abstraites

Une classe abstraite est une classe préfixée par le modificateur **abstract**

Une classe **abstraite** est **une classe n'ayant pas d'instances**.

Une classe abstraite ne peut pas être instanciée directement, et doit toujours être dérivée pour pouvoir générer des objets.

Les méthodes abstraites

Une méthode abstraite **n'est pas implémentée**, **elle n'a pas de corps**, **elle se réduit à sa signature suivie de «;»**.

Une méthode abstraite est une méthode dont le code doit être défini au niveau des classes filles

Remarque: Seules les méthodes d'instance peuvent être déclarées abstraites. (et non les méthodes de classes).

Dès qu'une classe contient au moins une méthode abstraite, elle doit être déclarée elle-même abstraite.

Les méthodes abstraites: Exemple

Elaborons une classe Vehicule comme abstraction des divers types de véhicules :

```
abstract public class Vehicule{
        protected String milieuDeDeplacement ;
        protected double poids ;
        protected double chargeUtile ;
        protected int nombreDePlaces ;
       abstract avancer(int nbreUnitesDepl) ;
       abstract reculer(int nbreUnitesDepl) ;
       abstract accelerer(int nbreUnitesAcceleration) ;
       abstract ralentir (int nbreUnitesAcceleration) ;
```

Cette classe est très générale, **les avions**, **les automobiles**, **les vélos**, **les navires**, **les trains** peuvent être des sous-classes de cette classe,

Les interfaces

Les interfaces

Une interface est **une collection de déclarations de méthodes dépourvues d'implémentation**. **Le code doit obligatoirement être précisé dans la redéfinition des méthodes de l'interface.**

Les interfaces

Exemple 1 :

Voler peut s'appliquer à des objets aussi différents qu'un avion, un oiseau, une feuille d'arbre…mais chacun de ces objets a une façon et des conditions différentes pour réaliser ce comportement.

Nous pourrions par exemple définir une **interface Vol** qui spécifie tout ce qui semble nécessaire comme comportement pour déclarer qu'il y a « vol » :

Les interfaces

```
public interface Vol {
public void monter(double hauteur) ;
public void descendre(double hauteur) ;
public void accelerer(double quantite) ;
public void ralentir(double quantite) ;
public void atterrir(double[] position) ;
public void decoller(double[] position) ;
public double indiquerVitesse(double temps) ;
public void tomber() ;
 }
```

si la classe Avion déclare implémenter l'interface Vol, elle devra fournir un corps à chaque méthode spécifiée dans l'interface Vol.

Les interfaces

Les champs déclarés dans le corps d'une interface sont implicitement déclarés **public**, **static** et **final** (c'est pourquoi, par abus de langage, on les qualifie de « constantes »):

public static final double PI = 3.141592653589793;

Il est préférable (pour des raisons de clarté) de placer les modificateurs public static et final devant les variables définies dans une interface mais cela n'est pas nécessaire.

Du fait que les champs sont implicitement qualifiés de final, **il est nécessaire qu'ils reçoivent une initialisation en même temps que leur déclaration**.

Les interfaces

On peut par exemple utiliser des interfaces pour « importer » des constantes :

```
public interface ConstantesMathematiques {
public static final double PI = 3.141592653589793;
public static final double logNep = 2.718281828459045;
public static final double gamma = 1.128787029908125;
public static final double fibonacci = 1.226742010720353;
public static final double racDeDeux = 1.414213562373095;
public static final double racDeTrois = 1.732050807568877;
public static final double logDeDix = 2.302585092994045;
public static final double Weierstrass =.474949379879206;
}
```

Les interfaces

Dans l'exemple qui précède on pourrait donc parfaitement écrire :

```
public interface ConstantesMathematiques {
    double PI = 3.141592653589793;
    double logNep = 2.718281828459045;
    double gamma = 1.128787029908125;
     ……
}
```

Constante classe

Ce sont des cas particuliers de variables de classe, dont **les valeurs ne doivent pas être changées**. On les déclare en combinant les modificateurs **final** avec **static**.

Par exemple, dans la classe **Math**, Math.PI est une constante définie par **public static final double PI**

```
// exemples de définitions de constantes
static final double charge_elem = 1.6E-19 ;
static final double nbAvogadro = 6.02E23 ;
double doublePI = 2* Math.PI ;
```

Le polymorphisme

Le polymorphisme

Le polymorphisme est un **concept fondamental** de la programmation orientée objet.

Dans la langue grec, il signifie « **peut prendre plusieurs formes** »

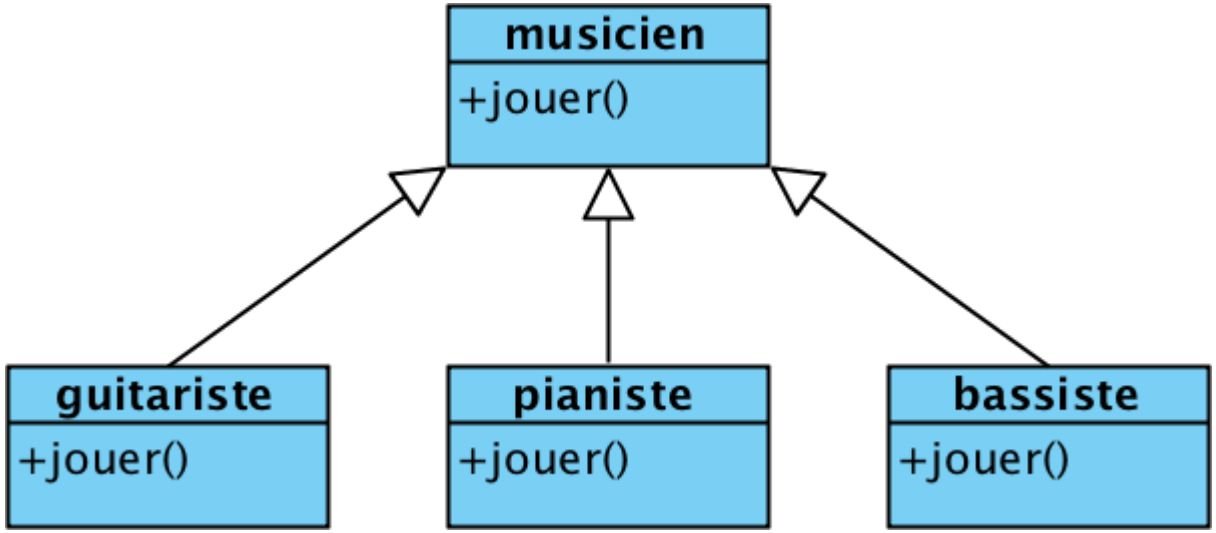
Le polymorphisme

Le polymorphisme est l’utilisation de la classe parente pour référencer une classe fille

Le polymorphisme : est la possibilité de choisir le code correct de la méthode à exécuter parmi un ensemble de méthodes redéfinies, en fonction de l'objet auquel s'adresse la méthode.

Le polymorphisme

La méthode "jouer" est présente dans la classe mère "musicien" et dans ses classes filles "guitariste", "pianiste" et "bassiste" :



L’appel de la méthode "jouer" sur tous les objets héritant de la classe "musicien" produira alors un résultat différent selon la sous classe à laquelle ils appartiennent.

Le polymorphisme

Exemple

```
Musicien [] musiciens = new Musicien [3];
musiciens[0] = new Guitariste (...);
musiciens[1] = new Pianiste(...) ;
musiciens[2] = new Bassiste(...);

musiciens[0].jouer() ;
musiciens[1]. jouer() ;
musiciens[2].jouer();
```