JAVA

Introduction à la programmation objet avec java – 2/4

SOMMAIRE

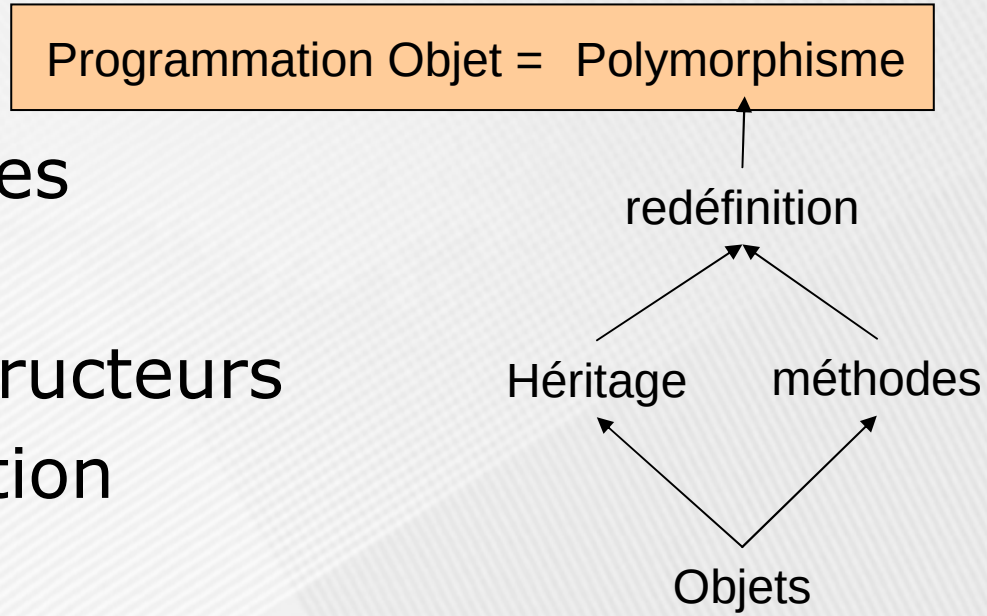
- Eléments de POO en Java (80mn)
- Mots clefs spéciaux (20mn)

Partie 1

POO en java

Notions dans le monde objets

- Objets & Classe
- propriétés & méthodes
- Encapsulation
- constructeurs & destructeurs
- surcharge & redéfinition
- Héritage
- Polymorphisme



Si vous savez déjà à quoi correspondent ces termes, vous pouvez quitter le cours **;-)** !

Introduction

Programmation structurée

- C, Pascal, ...
- **Programme** : procédures – fonctions
- **Question**: Que doit faire mon programme ?
  - structure !
  - traitements

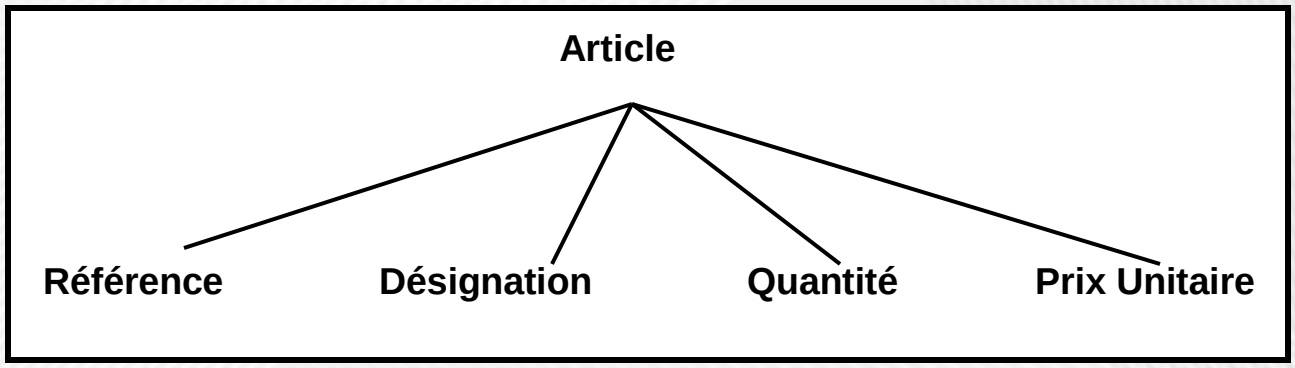
≠

Programmation objet

- Java, C++, ...
- **Question**: Sur quoi porte mon programme ?
  - les entités manipulées

Principe des modèles objets

- Les modèles à objets ont été créé pour modéliser le monde réel
  - Toute entité du monde réel est un objet
  - Tout objet représente une entité du monde réel
- un exemple : *Article*



Objet

objet
=
données + opérations sur ces données (méthodes)
=
variable de "type abstrait" (non scalaire)
=
entité du domaine du problème

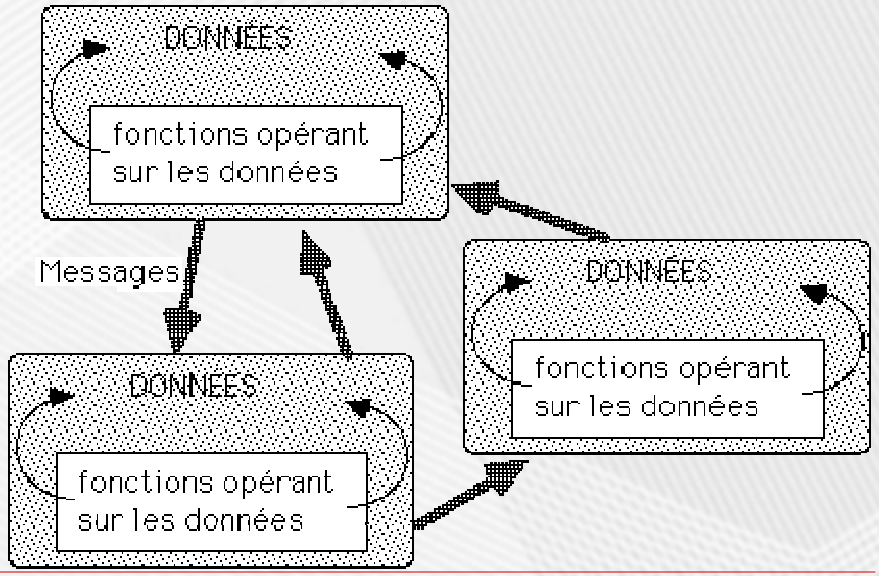
Exemple d'objets : des voitures

```
objet R21_de_mon_chef
   genre : Renault
   plaque : 2245 CDV 75
   NbPlaces : 5
   propriétaire : chef
   s_arreter()
   avancer()
fin objet
```

```
objet ma_Clio
   genre : Renault
   plaque : 4357 NBG 93
   NbPlaces : 4
   propriétaire: Moi
   s_arreter()
   avancer()
fin objet
```

Programme objet

- Un programme = une société d'entités
- Son exécution : les entités s'entraident pour résoudre le problème final en s'envoyant des messages.
- une entité = un objet qui prend en compte sa propre gestion (objet responsable)
- un message = est traduit en appel de méthode



Principe d'encapsulation

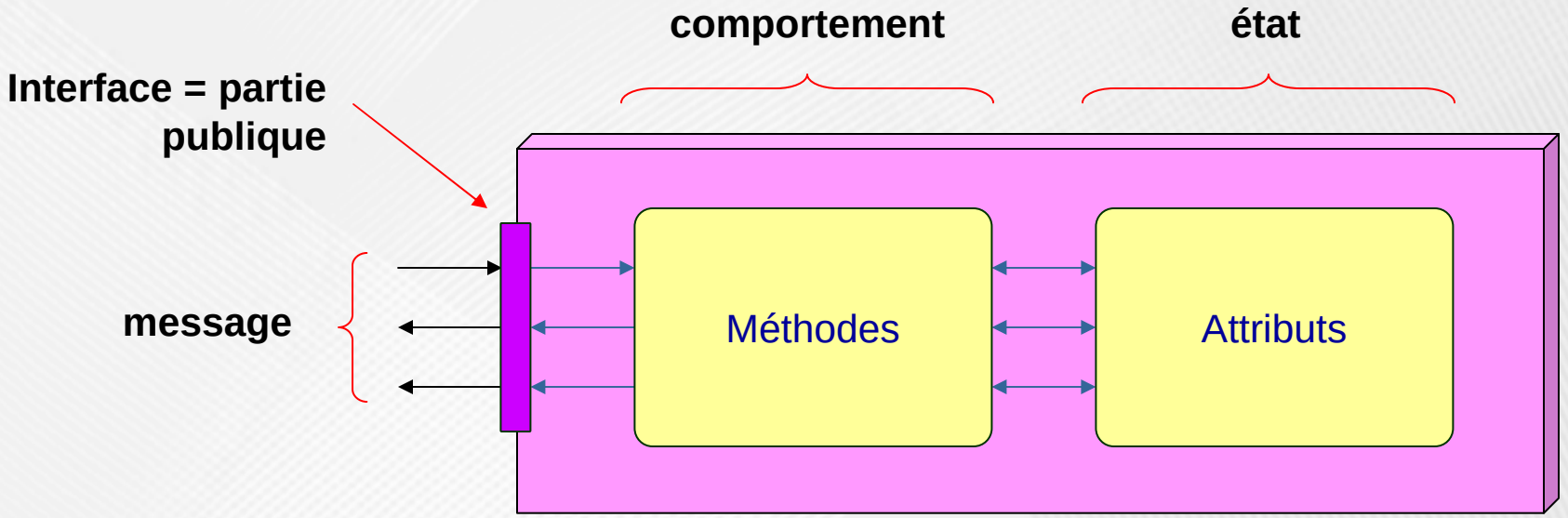
Encapsulation
=
regroupement de code et de données

masquage d'information au monde extérieur (data hiding)

- Un objet est composé de 2 parties :
  - partie publique : opérations qu'on peut faire dessus
  - partie privée : partie interne (intime) = données sensibles de l'objet (les attributs et les autres méthodes)
- Les utilisateurs de l'objet ne voient (cad ne peuvent utiliser et ceci est contrôlé par le compilateur) que la partie publique (qui est un ensemble de méthodes)
- Objectifs :
  - Seul l'objet peut modifier ses propre attributs
  - Masquer les détails de l'implémentation

Principe d'encapsulation

- Objectif :



- Catégories d'autorisation d'accès :
  - public : aucune restriction
  - private & protected : accès réservé (private est plus restrictive que protected)

Classe

classe
=
modèle décrivant le *contenu* et le *comportement* des futurs objets de la classe
=
ensemble d'objets

le contenu = les données
le comportement = les méthodes

Exemple: la classe des véhicules, la classe des camions, des automobiles.
La classe des automobiles peut être décrite par :

```
classe Automobile
   genre
   plaque
   NbPlaces
   propriétaire
   s_arreter()
   avancer()
fin classe
```

Résumé :classe & objet

- classe & objet, méthode et message :
  - Un exemplaire particulier d'une classe s'appelle une instance de la classe ou un objet de cette classe : objet = instance de classe
  - Une classe est un agrégat d'attributs et de méthodes : les *membres*
  - En première approche:
    - les objets sont à la programmation objet ce que sont les variables à la programmation structurée
    - Les classes sont à la programmation objet ce que les types sont à la programmation structurée

programmation procédurale	Variable	type
Programmation objet	Objet	classe

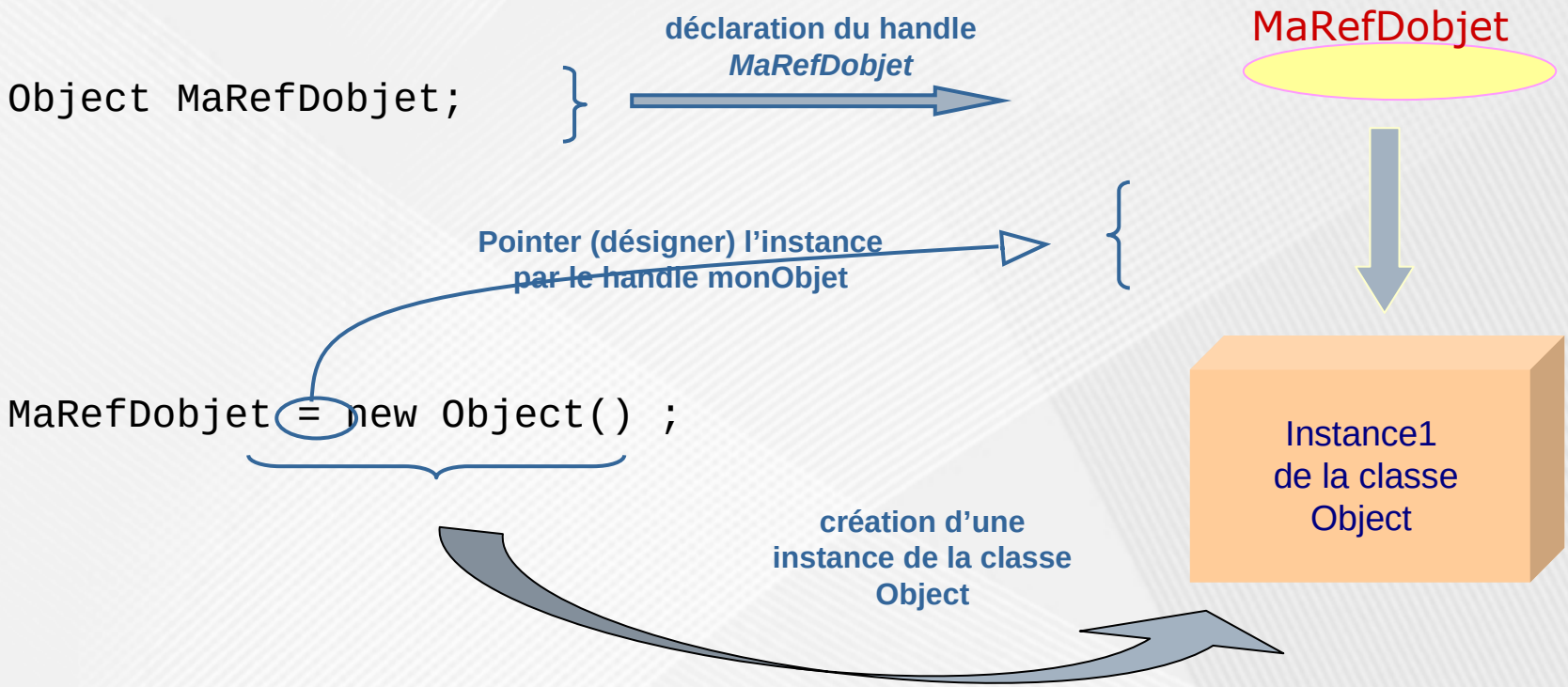
  - Envoyer un message à un objet c'est lui demander d'exécuter une de ses méthodes.

Classes et objets : exemple

Propriété / méthode
référence d'un objet :`Cercle c;`
créer un objet: l'opérateur new
constructeur

```
class Cercle {
    double x, y;
    double r;

    Cercle(double R) {
        r = R;
    }
    double aire() {
        return 3.14159 * r * r;
    }
}

class MonPremierProgramme
{
    public static void main(String[] args) {
        Cercle c;           // c est une référence sur un objet Cercle, pas un objet
        c = new Cercle(5.0); // c référence maintenant un objet alloué en mémoire

        c.x = c.y = 10;
        System.out.println("Aire de c :" + c.aire());
    }
}
```

Objets et références



Syntaxe : classes & objets

CLASSE

- tout est défini à l'intérieur de classe : il n'y a rien de global
  - Les méthodes sont définies directement dans la classe
  - les constantes aussi
- On définit les classes par :

```
class nomClasse {
     type_donnees données
     définition des méthodes
}
```

- par exemple :

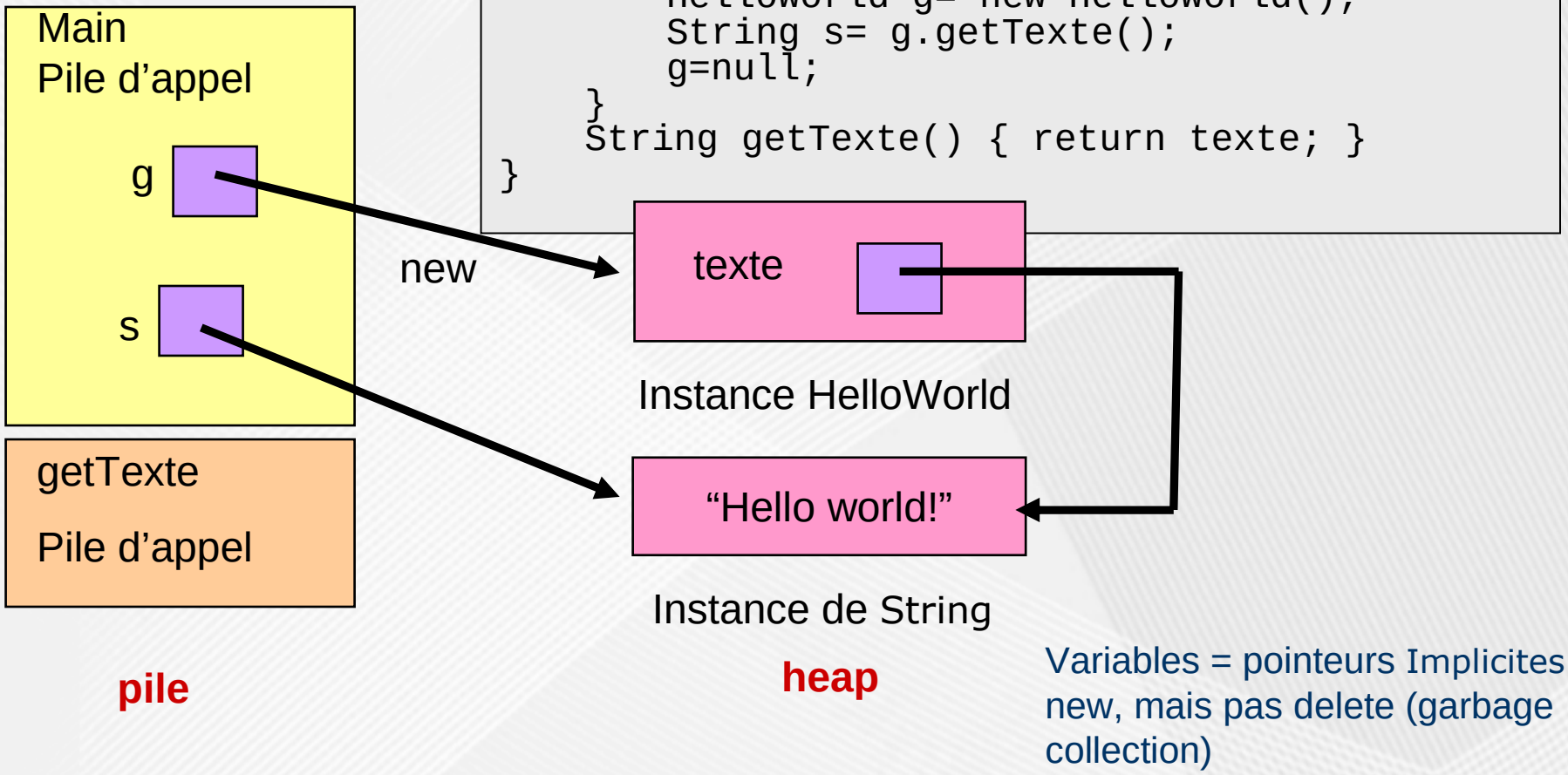
```
class Automobile {
   String genre;
   String immatriculation;
   int NbPlaces;
   String propriétaire;
   void s_arreter(){ ... }
   void avancer(float nbmetres)
    { ... }
}
```

OBJET

- En Java il faut construire explicitement les objets (**new**)
- Par la suite les objets construits seront repérés et accessibles grâce à leur référence initialisée au moment de la construction de l'objet.
- `On a donc :`
  `Automobile ma_Clio;`
  déclaration d'une référence.
- Puis
  `ma_Clio = new Automobile();`
- Ayant une référence `ma_Clio` initialisée, on l'utilise alors pour lancer toute méthode sur l'objet (i.e. pour envoyer tout message à l'objet) par :
  `ma_Clio.avancer(4.5);`

Classes et objets: java memory model

```
Class HelloWorld {
    String texte="Hello world!";
    public static void main (String args[]) {
        HelloWorld g= new HelloWorld();
        String s= g.getTexte();
        g=null;
    }
    String getTexte() { return texte; }
}
```



Constructeur

- Pour initialiser un objet on utilise une méthode "spéciale" appelée constructeur (même nom que la classe sans valeur de retour)
- elle est appelé automatiquement au moment du **new**

```
class Cercle {
    double x, y, r;
    Cercle(double X, double Y, double R) {
        x = X; y = Y; r = R;
    }
    Cercle(Cercle c) {
        x = c.x; y=c.y; r=c.r;
    }
    Cercle() {
         x=y=r=0;
    }
}
```

Un constructeur « par copie »

- Toute classe possède un constructeur par défaut, implicite.
- Celui-ci peut être redéfini (ici le dernier constructeur)
- Une classe peut avoir plusieurs constructeurs qui diffèrent par le nombre et la nature de leurs paramètres **(surcharge)**

Destruction d'un objet (1/2)

- La destruction des objets est prise en charge par le *garbage collector (GC).*
- Le GC détruit les objets pour lesquels il n'existe plus de référence.
- Les destructions sont asynchrones (le GC est géré dans un *thread* de basse priorité).
- Aucune garantie n'est apportée quant à la destruction d'un objet.
- Si l'objet possède la méthode finalize, celle-ci est appelée lorsque l'objet est détruit.

Destruction d'un objet (2/2)

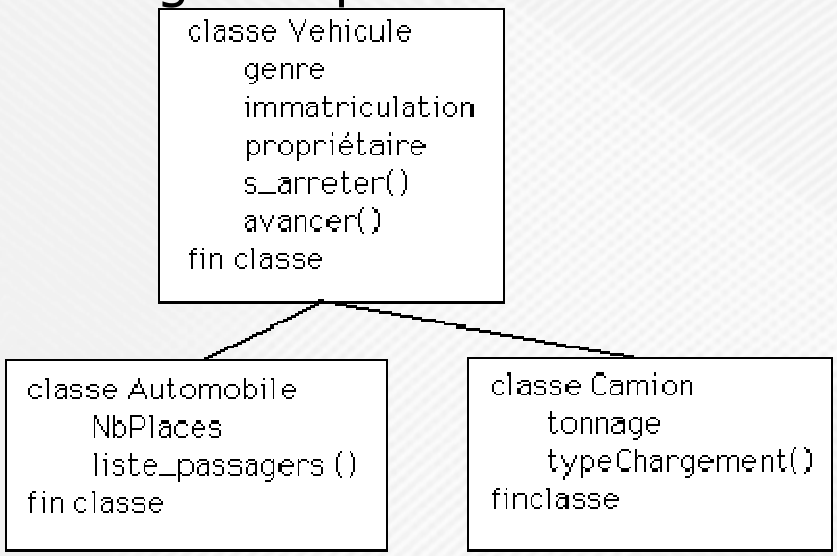
```
public class Cercle {
    ...
    void finalize() { System.out.println("Je suis garbage collecte"); }
}
...
    Cercle c1;
    if (condition) {
        Cercle c2 = new Cercle(); // c2 référence une nouvelle instance
        c1 = c2;
    }
    // La référence c2 n'est plus valide mais il reste une référence,c1,
  // sur l'instance

    c1=null; // L'instance ne possède plus de référence. Elle n'est plus
           // accessible.
... // A tout moment le gc peut detruire l'objet.
    System.gc();
```

Héritage (1/4)

Héritage
=
construire une classe à partir d'une autre
=

Notion rattachée aux classes

héritage simple:



Automobile et Camion héritent (ou dérivent) de Vehicule.

héritage = dérivation

- La classe dont on dérive est dite classe de base (ou parente ou mère)
- Les classes obtenues par dérivation sont dites classes dérivées.
- L'héritage permet de reprendre intégralement tout ce qui a déjà été fait et de pouvoir l'enrichir (réutilisabilité)
- en dérivant, on établit ainsi une spécialisation

Héritage (2/4)

- Une sémantique forte :
  - Lorsqu'une classe B hérite d'une classe A, les objets de B seront considérés aussi comme une "sorte" d'objet A (IMPORTANT)
  - L'héritage entre classes traduit la sémantique **"est une sorte de"** :
    - une automobile "est une sorte de " véhicule
    - donc la classe automobile hérite de la classe Vehicule

- syntaxe :
  - Java utilise le mot clé réservé `extends` :

```
class parente // ou classe mère
{
        // Attributs et méthodes
}

class SousClasse extends parente
      // ou classe fille
{
        // Attributs supplémentaires
        // Méthodes propre à la classe
}
```

  - par exemple:

```
class Camion extends Vehicule {
       double tonnage;
       String typeChargement()

         { … }
     }
```

Héritage (3/4)

- Une classe ne peut hériter (extends) que d'une seule classe à la fois.
- Les classes dérivent toujours, par défaut, de java.lang.Object
- Une référence d'une classe C peut contenir des instances de C ou des classes dérivées de C.
- L'opérateur instanceof permet de déterminer la classe d'une instance.

Héritage (4/4)

```
class Ellipse {
     double r1, r2;
     Ellipse(double r1, double r2) { this.r1 = r1; this.r2 = r2;)
     double aire{...}
}

class Cercle extends Ellipse {
     Cercle(double r) {r1=r; r2=r;}
     double getRayon() {return r1;}
}

Ellipse e = new Ellipse(2.0, 4.0);
Cercle c = new Cercle(2.0);

System.out.println((e instanceof Cercle));  // false
System.out.println((e instanceof Ellipse)); // true
System.out.println((c instanceof Cercle));  // true
System.out.println((c instanceof Ellipse)); // true (car Cercle dérive de Ellispe)
e = c;
System.out.println((e instanceof Cercle));  // true
System.out.println((e instanceof Ellipse)); // true
int r = e.getRayon();    // Error: method getRayon not found in class Ellipse.
c = e;                    // Error: Incompatible type for =. Explicit cast needed.

int r = ((Cercle)e).getRayon();    // OK
c = (Cercle)e;                       //OK
```

Le masquage des variables

- Une classe peut définir des variables portant le même nom que celles de ses classes ancêtres
- Une classe peut accéder aux attributs redéfinis de sa classe mère en utilisant super ou par cast.
- Le mot clef super fait référence à la classe parente *(plus de détail à la fin de ce cours)*

Spécialisation des méthodes

- On considère 2 classes liées par l'héritage.
  Par exemple :

```
class Employee { . . . }
class Manager extends Employee { . . . }
```

- Souvent une méthode est définie dans la classe de base, mais a besoin d'être "affinée" dans la classe dérivée. On a par exemple :

```
class Employee {
   long calculPrime() { return 1000; } // une prime pour un employé
}

class Manager extends Employee {
   long calculPrime()
    { return 5000; } // une prime + grosse pour un manager
}
```

- Les 2 méthodes `calculPrime()` de ces 2 classes ont même signature. On dit que la méthode `calculPrime()` de la classe `Manager` spécialise la méthode `calculPrime()` de la classe `Employee`.

- **Autre terminologie:** **Spécialisation = Redéfinition**

Polymorphisme

- Le nom de la méthode `calculPrime()` désigne 2 codes distincts. Autrement dit l'identificateur `calculPrime` représente "plusieurs formes" de code distinct. On parle de polymorphisme.
- On utilise le polymorphisme à l'aide d'une référence d'objet de classe de base. Par exemple on écrit en Java :

```
Employee e = new Employee();
p=e.calculPrime(); // lance la méthode calculPrime() de la classe Employee
```

- mais on peut écrire aussi:

```
e = new Manager();
p=e.calculPrime(); // lance la méthode calculPrime() de la classe Manager
```

- Un des avantages de ceci est de pouvoir utiliser une seule référence e de la classe `Employee` pour référencer un `Employee` ou un `Manager`. C'est possible car un `Manager` (classe dérivée de la classe `Employee`) est *une sorte de* `Employee` : sémantique de l'héritage.
- Autre exemple : si une automobile "est une" sorte de véhicule, alors la classe automobile hérite de la classe véhicule, et une référence de type véhicule peut contenir des instances de véhicule ET de automobile

Règle

Une référence d'objet de classe de base(mère) peut **aussi** contenir des instances des classes filles

Polymorphisme = liaison dynamique

- Le gros avantage du polymorphisme est de pouvoir référencer des objets sans connaître véritablement leur classe (mais celle d'une classe mère), et de pouvoir par la suite, lancer le code approprié à cet objet au moment de l'exécution du programme (et non pas à la compilation).
- On peut ainsi écrire un code comme par exemple

```
int calculSalaire(Employe e)
{
    int salaire = e.calculSalaireDeBase();
    salaire += e.calculPrime();
    return salaire;
}
```

- Au moment de la compilation, le compilateur ne sait pas si c'est la méthode `calculPrime()` de la classe `Employee` ou celle de la classe `Manager` qui sera lancée.
- Autrement dit, la liaison (l'opération de « link ») entre l'identificateur `calculPrime()` et le code à lancer est déterminé à l'exécution du programme : la liaison est dynamique

Polymorphisme (3)

Reprenons l'exemple précédent:

```
(1)Employee e = new Employee();
(2)e.calculPrime(); // lance la méthode calculPrime() de la classe Employee
(3)e = new Manager();
(4)e.calculPrime(); // lance la méthode calculPrime() de la classe Manager
```

- Remarquer que si e référence un `Manager(ligne 3)`, en ligne 4 le code de la méthode `e.calculPrime()` de la classe `Employee` ne sera **pas** executé
  - **Le code à lancer est déterminé suivant la classe de l'objet et non pas la classe de la référence à cet objet**

Comment cela fonctionne t il ?

- un appel comme : `e.calculPrime()` recherche au moment de l'exécution l'objet référencé par e
- A partir de cet objet, l'interpréteur cherche si cette méthode `calculPrime()` existe dans la classe de cet objet. Si c'est le cas, cette méthode est lancée et le processus est terminé.
- Si ce n'est pas le cas, l'interpréteur refait cette recherche dans la classe de base, et ainsi de suite
- une exception peut être levée dans des cas (tordus) où une telle méthode n'est pas trouvée les classes parentes
- En conclusion il est fondamental de voir que pour utiliser le polymorphisme, il faut :
  - avoir au départ une arborescence d'héritage sur les classes en jeu
  - utiliser des références d'objet de la classe de base (la classe mère)
  - avoir (au moins) deux méthodes de même signature définies l'une dans la classe de base, l'autre dans la classe dérivée.

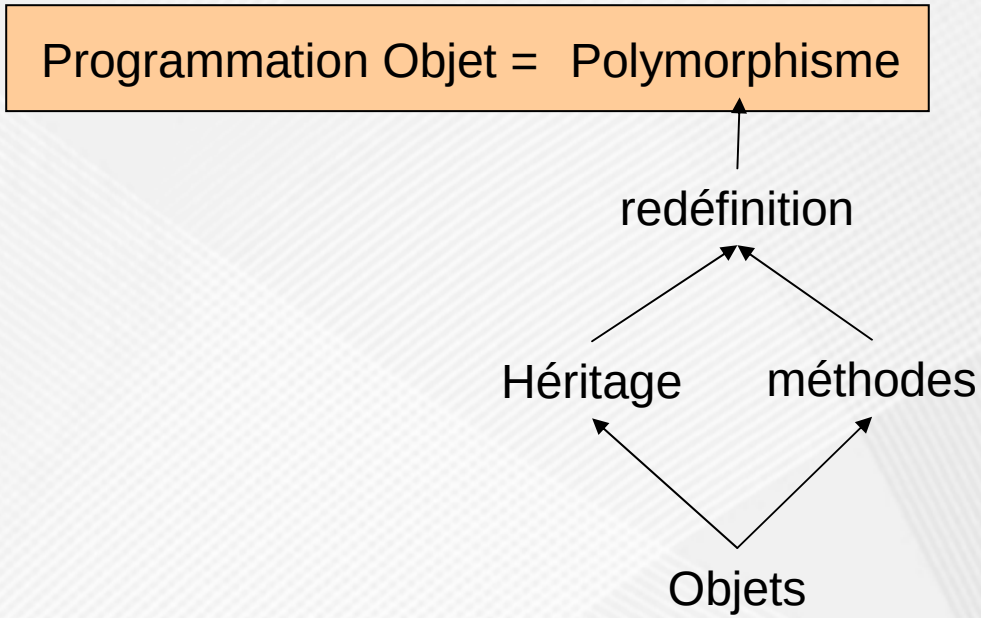
Surcharge vs. polymorphisme

- Il ne faut pas confondre les notions de surcharge et polymorphisme (ou liaison dynamique) qui ont finalement peu de point commun.
- Voici un tableau qui résume les différences :

	Surcharge	**Polymorphisme**
Héritage	nul besoin	nécessite une arborescence de classes
signature des méthodes	doivent différer	doivent être les mêmes
résolu à	la compilation	l'exécution

Résumé sur la POO

- Cela reprend le plan du cours:



Partie 2

Mots clefs spéciaux

Mot clef this

- Le mot clef this fait référence à l'instance en cours quand on est dans le code de la classe.
  - il permet de différencier les attributs d'une classe des paramètres passées à une méthode
  - Exemple:

```
class Cercle {
double r;
Cercle(double r) {
        this.r = r;
}
...
```

- Il peut être aussi utilisé pour invoquer un constructeur de la même classe
  - Exemple :

```
public Cercle () {
        this ( 0.0 ) ;
}
```

Cet ligne fait appel au constructeur ci-dessus

- Note: les méthodes statiques ne peuvent pas accéder à **this**
  - Revoir le cours 1 pour le mot clé static

Mot clef super

- Vous savez que grâce à l'héritage:
  - Une classe peut définir des variables portant le même nom que celles de ses classes ancêtres (masquage)
  - Une classe peut définir des méthodes portant le même nom que celles dans sa classe mère (redéfinition)
- Le mot clef super est un préfixe qui fait référence à la classe parente
  - *Grace à super, une classe peut accéder aux attributs et méthodes de sa classe mère*

Mot clef super: exemple

```
class A {
    int x=1;
    void m() {...}
}
class B extends A{
    int x=2;
    void m() {...}
}
class C extends B {
    int x=3, a;
    void m() {...}
    void test() {
        a = x;
        a = super.x;          // a reçoit la valeur de la variable x de la classe B
        a = super.super.x;    // Syntax error
        a = ((A)this).x;      // a reçoit la valeur de la variable x de la classe A
        a = ((B)this).x;      // a reçoit la valeur de la variable x de la classe B
        super.m();            // Appel à la méthode m de la classe B
        super.super.m();      // Syntax error
        ((A)this).m();        // Appel à la méthode m de la classe A (et non C)
    }
}
```

Mot clef super

- Le mot clef `super` permet d'appeler le constructeur (tous les types de constructeur) de la classe mère
  - Exemple :

```
public class Ellipse {
   double r1, r2;
   Ellipse(double r1, double r2)
   { this.r1 = r1; this.r2 = r2;)
}

final class Cercle extends Ellipse {
   int color;
   Cercle(double r) {super(r, r); color=0; }
}
```

  - le constructeur de Cercle fait appel au constructeur parent plutôt que d 'initialiser lui-même r1 et r2
  - il peut ensuite faire tout autre initialisation qui lui est propre (`color=0;`)
  - Remarque: le mot clef `super` doit apparaître à la première ligne du constructeur.

Mot clef abstract

- `abstract`: Mot clef qui s'applique à une classe ou une méthode
- une méthode est dite abstraite si on ne veut écrire que le prototype (=la « signature ») - pas le corps
- Si dans une classe, au moins une méthode est déclarée abstract, alors la classe le devient
- L'implémentation (=le corps) de la méthode s'effectuent dans les classes qui héritent de cette classe
- Pour ne plus être abstraite, une classe fille doit implémenter toutes les méthodes abstraites de la classe dont elle hérite
- Si une classe est déclarée abstract, alors elle ne peut pas être instanciée (pas de new possible)

```
abstract class C1 {
    abstract void toto( );
    void normale( ){
      // corps de la méthode
   }
}
```

```
class C2 extends C1{
   void toto( ){
      // corps de la méthode
   }
}
```

Mot clef final

- `final` : Mot clef qui s'applique à une classe, méthode ou propriété.
  - Classe: `final` indique que la classe ne peut plus être héritée
  - Méthode: `final` indique que la méthode ne peut redéfinie dans les classes dérivées.
    - Une méthode `static` ou `private` est automatiquement final
  - propriété: `final` indique qu'il s'agit d'une constante. Une propriété final ne peut être affectée qu'une seule fois

Mot clef static

- `static` : Mot clef réservé aux attributs et méthodes (attribut ou méthode de classe) :
  - Un attribut statique existe dès que sa classe est invoquée, en dehors de toute instanciation. Il existe en un seul exemplaire (~ variable globale)
  - Une méthode statique ne manipule pas des attributs de sa classe. Elle rend ainsi un service global.
  - De l'extérieur d'une classe une méthode ou un attribut statique s'emploie de la manière suivante :

    ```
    Nom_classe.nom_statique
    ```

4 - Mots clefs : résumé

- Une méthode peut :
  - Appartenir à la classe (static)
  - Appartenir aux instances (*sans* static)
  - Ne pas avoir d'implémentation (abstract)
  - Bloquer toute surcharge (final)
- Une classe peut :
  - être non instanciable (abstract)
  - Bloquer tout héritage (final)