

Introduction à la programmation en JAVA



Table des matières

- I. Introduction à Java et historique du langage
- II. Notre outil de développement : *Eclipse Mars*
- III. Le langage Java et sa syntaxe
- **IV. La POO avec Java**
- V. API Java
- VI. La gestion des exceptions
- VII. Les collections
- VIII. La sérialisation

Première approche du concept de POO



Aperçu du chapitre

I. POO <> Procédural

II. Penser le monde en objets

III. Le concept de classe

IV. La notion de package

V. L'encapsulation

VI. Les associations

VII. L'héritage

VIII. Le polymorphisme

IX. Les classes abstraites et les interfaces

X. Les classes internes

I . POO <> Procédural

La programmation procédurale (ou modulaire) :

La programmation procédurale consiste à découper un programme en une **série de fonctions** (ou procédures) : c'est ce qu'on appelle l'abstraction procédurale.

Ces fonctions contiennent un certain nombre d'instructions qui ont pour but de réaliser un traitement particulier (calcul de la circonférence d'un cercle, impression de la fiche de paie d'un salarié, ...).

Dans le cas de l'approche procédurale, un programme correspond à **l'assemblage de plusieurs fonctions** qui s'appellent entre elles.

I . POO <> Procédural

Les inconvénients de la programmation procédurale :

- Ne tient pas compte de la structure des données d'un programme (pas de modèle de données)
- Difficulté de réutilisation du code dans un autre projet
- Séparation entre les données et leurs traitements inexistante
- Code peut devenir difficilement maintenable, lisible,...
- Suite d'instructions s'exécutant de façon linéaire

I . POO <> Procédural

La programmation orientée objet:

La programmation orientée objet (POO) ou programmation par objet, est un paradigme de programmation informatique qui consiste en la **définition et l'assemblage de briques logicielles appelées objets**.

Un **objet** représente un **concept**, une **idée** ou toute **entité** du monde physique.

Une **application** est alors vue comme un **ensemble d'objets** qui interagissent entre eux au moyen de **messages**.

Un programme qui utilise l'approche objet s'appuie sur trois techniques fondamentales qui seront présentées plus tard en détails :

- L'encapsulation
- L'héritage
- Le polymorphisme

Aperçu du chapitre

I. POO <> Procédural

II. Penser le monde en objets

III. Le concept de classe

IV. La notion de package

V. L'encapsulation

VI. Les associations

VII. L'héritage

VIII. Le polymorphisme

IX. Les classes abstraites et les interfaces

X. Les classes internes

II . Penser le monde en objets

Chaque objet est composé de 2 parties:

1. Partie **statique** qui décrit l'**état** de l'objet
2. Partie **dynamique** qui détermine le **comportement** de l'objet

L'**état** d'un objet en POO est l'ensemble des valeurs des données (ou attributs) de l'objet à un moment donné.

Les attributs de l'objet sont représentées par des variables.

Cycle de vie d'un objet :

création → succession de changements d'état → destruction

II . Penser le monde en objets

Dès qu'un objet ne possède plus de référence en mémoire (n'est plus utilisé), cet objet sera « ramassé » par le **garbage collector**.

Le garbage collector (ou ramasse-miettes) est un outil utilisé pour la gestion de la mémoire lors de l'exécution.

Ainsi, contrairement à d'autres langages comme le C++, le développeur ne doit pas s'occuper de la destruction des objets.

Le principe de base de la récupération automatique de la mémoire est simple :

- Déterminer, au cours de l'exécution, quels objets dans le programme ne sont plus utilisés
- Libérer la mémoire utilisée par ces objets.

II . Penser le monde en objets

Le **comportement** d'un objet est représenté par ses **méthodes** et les messages qu'elles peuvent traiter.

Ces méthodes sont capables de modifier l'état de l'objet auquel elles appartiennent.

Exemple : une voiture est caractérisée par sa couleur, sa consommation, sa vitesse et par ses actions (freiner, accélérer, ...).

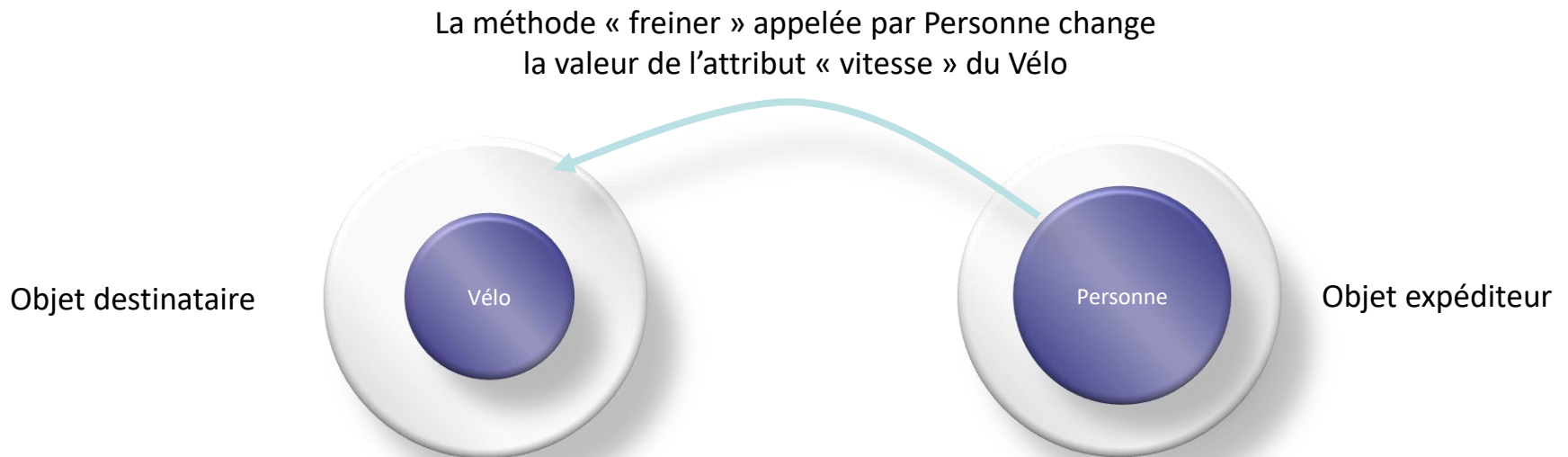
Une méthode est composée d'une en-tête comprenant, entre autres, son identificateur et d'un bloc d'instructions qu'elle exécute à chaque fois qu'elle est appelée.

II . Penser le monde en objets - *Interaction entre objets via des messages*

Une application OO comporte le plus souvent de 1 à n objets.

Les objets interagissent entre eux par le biais de messages qu'ils s'envoient.

Quand un objet, dans une de ses méthodes, déclenche une méthode sur un autre objet, un message part d'un objet expéditeur vers un objet destinataire.



Aperçu du chapitre

I. POO <> Procédural

II. Penser le monde en objets

III. Le concept de classe

IV. La notion de package

V. L'encapsulation

VI. Les associations

VII. L'héritage

VIII. Le polymorphisme

IX. Les classes abstraites et les interfaces

X. Les classes internes

III . Le concept de classe

L'**abstraction** consiste à partir d'un **ensemble d'objets** et en trouver les **caractéristiques communes** (attributs et méthodes) pour en **créer une classe**.

Tout objet doit se conformer à ce qui est défini dans la classe.

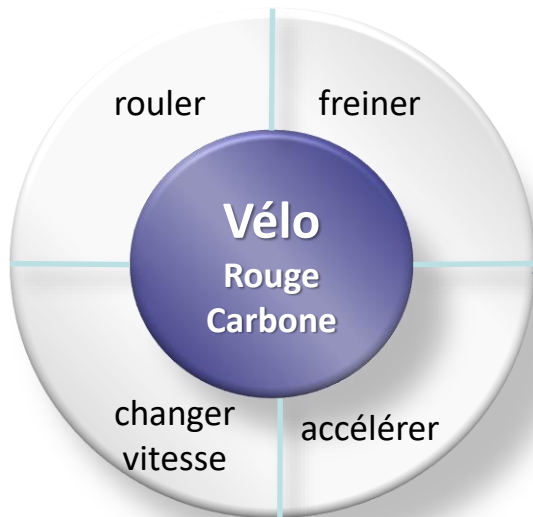
L'**instanciation** d'une classe consiste à créer un objet à partir de celle-ci.

Définitions:

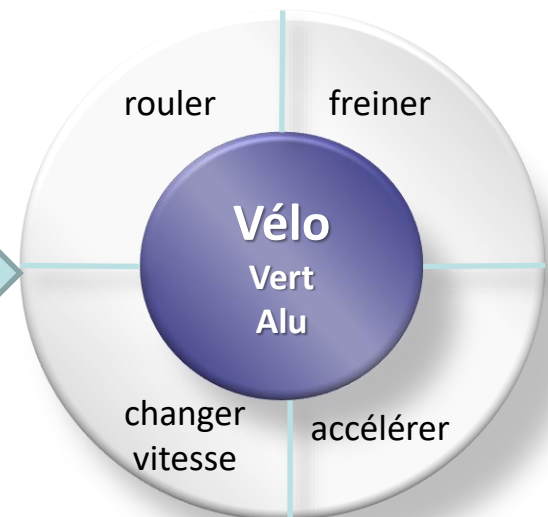
- Classe : type de l'objet
- Objet : instance d'une classe
- Méthode : ensemble d'instructions pour agir sur les attributs de l'objet
- Attribut : variable définissant l'objet, accessible à toutes les méthodes de la classe

III . Le concept de classe

Classe Vélo



La classe vélo
joue le rôle
d'un usine.
Chaque vélo
créé avec
cette classe
est une
instance de
celle-ci



III . Le concept de classe

Si un objet est une instance d'une classe, alors on peut considérer une classe comme une empreinte, un prototype, une usine, un moule qui définit les attributs et les méthodes communs à tous les objets issus de cette classe.

Ainsi, après avoir créé la classe Vélo, il faut donc instancier (créer une instance) avant de pouvoir l'utiliser.

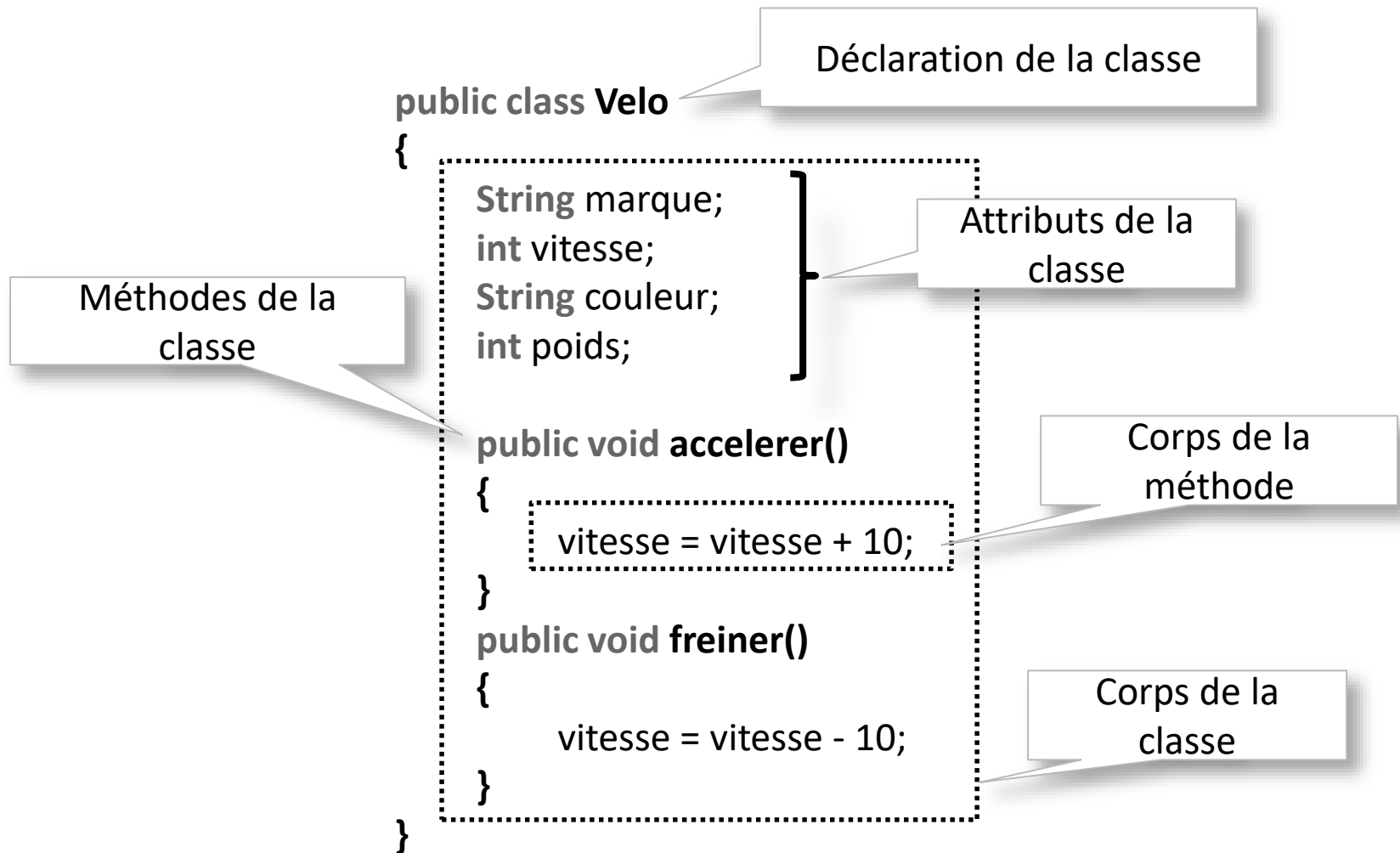
Classe Vélo



```
public class Velo
{
    String marque;
    int vitesse;
    String couleur;
    int poids;

    public void accelerer()
    {
        vitesse = vitesse + 10;
    }
    public void freiner()
    {
        vitesse = vitesse - 10;
    }
}
```


III . Le concept de classe – *Aperçu d'une classe*



III . Le concept de classe – *Les attributs*

```
public class Velo  
{  
    String marque;  
    int vitesse;  
    String couleur;  
    private int poids;  
  
    ...  
}
```



Attributs de la classe

Les **attributs** de la classe sont les **variables** déclarées dans le corps de la classe et **en dehors de toute méthode**.

Si on se réfère à la notion de portée d'une variable, elles sont bien accessibles n'importe où dans la classe.

III . Le concept de classe – *Les attributs*

Les **attributs** sont composés de 3 parties :

1. Modificateur d'accès :
 - `private` : accessible uniquement dans la classe dans laquelle l'attribut est déclaré
 - `public` : accessible dans toutes les classes
 - `protected` : accessible dans la classe dans laquelle l'attribut est déclaré, les classes enfants et les classes du même package
2. Type de l'attribut
3. Nom de l'attribut

Exemple:

```
private String nom;
```

III . Le concept de classe – *Les méthodes*

```
public class Velo  
{
```

```
...
```

```
public void accélérer()
```

```
{
```

```
    vitesse = vitesse + 10;
```

```
}
```

```
...
```

```
}
```

Déclaration de la
méthode

Corps de la méthode

Les méthodes de la classe sont composées d'un corps qui reprend les instructions et d'une partie déclarative (en-tête) qui comprend:

1. un modificateur d'accès
2. le type de retour
3. le nom
4. des paramètres en entrée (arguments)

III . Le concept de classe – *Les méthodes*

Exemple:

```
public boolean maMethode (int args1, double args2) {  
    ...  
}
```

Si une méthode ne **retourne aucune valeur**, on utilise le mot-clé **void** comme type de retour.

Dans le **cas inverse**, le type de retour est le type de la variable retournée.
Une variable est retournée à l'aide du mot-clé **return**.

Une méthode peut recevoir un ensemble d'**arguments** entre les parenthèses. Ceux-ci sont séparés par une virgule et utilisés lors de l'exécution de la méthode.

Dans le cas d'un argument de **type primitif**, c'est la **valeur** du type qui est copiée.
Dans le cas d'un argument de **type objet**, c'est la **référence** du type qui est copiée.

III . Le concept de classe – *Les méthodes*

La **signature** d'une méthode est constitué par le nom de la méthode ainsi que la liste des types des arguments de celle-ci.

Exemple :

```
public double calculerSalaire(double taux) { ... } ➔ calculerSalaire(double)
```

La signature est associée de manière **unique** au corps d'instructions qui compose la méthode.

Deux méthodes ne peuvent pas posséder la même signature. Sinon, il serait impossible de les identifier.

La **surcharge de méthodes** consiste à créer une nouvelle méthode portant le même nom qu'une autre méthode mais dont la signature se différencie par le nombre ou le type des arguments.

III . Le concept de classe – *Le constructeur*

Si une classe peut être vue comme une usine servant à créer des objets d'un certain type, alors le **constructeur** d'une classe correspond aux chaînes de montage de cette même usine.

Le constructeur d'une classe est une méthode particulière qui :

- Porte le même nom que la classe
- Ne possède pas de type de retour (même pas void!)
- A pour but d'initialiser les attributs de l'objet lors de sa création
- N'est appelé uniquement lors de la construction de l'objet
- Est, le plus souvent, public
- Peut être surchargée

Il existe toujours un **constructeur par défaut** si vous n'en définissez pas dans la classe. Il ne comporte aucun paramètre et laisse la valeur par défaut de chaque attribut.

III . Le concept de classe – *Le constructeur*

```
public class Velo  
{  
    String marque;  
    int vitesse;  
    String couleur;  
    int poids;
```

Constructeur

```
Velo monVelo = new Velo("VTT" , "rouge");
```

```
    public Velo (String a_marque, String a_couleur )  
    {  
        marque = a_marque;  
        couleur = a_couleur;  
    }  
}
```


III . Le concept de classe — *Le mot clé this*

Le mot-clé **this** permet d'accéder aux attributs et méthodes de la classe.

Facultatif, ce mot-clé sera par contre obligatoire pour différencier, par exemple, des attributs d'une classe qui auraient le même nom que les paramètres passés à une méthode ou un constructeur.

```
public class Velo
{
    String marque;
    String couleur;

    public Velo(String marque, String couleur)
    {
        this.marque = marque;
        this.couleur = couleur;
    }
}
```

III . Le concept de classe – *Variables de classe*

Les attributs vus jusqu'à présent sont appelés des **variables d'instance** et ont une valeur **propre à l'objet** auquel ils appartiennent.

Une **variable de classe** est un type d'attribut particulier dont la valeur est **partagée** par tous les objets de la classe.

Une variable de classe est déclarée à l'aide du mot-clé **static**.

A la différence des variables d'instances, les variables de classes sont souvent **initialisées à la déclaration** et non dans le constructeur,

Exemple:

```
private int nomClient; ➔ variable d'instance  
public static int nbClients = 0; ➔ variable de classe
```

III . Le concept de classe – *Méthodes de classe*

Les méthodes vues jusqu'à présent sont appelées des **méthodes d'instance** et s'exécutent sur un objet en utilisant les attributs liés à cet objet.

Une **méthode de classe** est un type de méthode particulier qui sert à rendre un service indépendamment de variables d'instance ou qui permet de manipuler des variables de classes.

Les données manipulées par ces méthodes doivent pouvoir exister sans objet !

Une méthode de classe est déclarée à l'aide du mot-clé **static** et peut être directement appelée sur le nom de la classe.

Exemple:

```
private static int getNbClients() { ... } ➔ déclaration  
Client.getNbClients(); ➔ appel
```

III . Le concept de classe – *Méthodes utilitaires*

Certaines méthodes, très pratiques, peuvent être définies dans chaque classe.

- `public String toString()`
renvoie l'état d'une instance de la classe. Si elle n'est pas définie, cette méthode retourne l'adresse de l'objet.
- `public typeObject clone()`
retourne une copie de l'objet en copiant la valeur des attributs de cet objet dans un autre nouvellement créé. Ainsi, l'objet retourné possède une référence (adresse) différente de celle de l'objet initial.
- `public boolean equals(Object o)`
permet la comparaison des objets en se basant sur la valeur de leurs attributs et non sur les adresses.

Aperçu du chapitre

I. POO <> Procédural

II. Penser le monde en objets

III. Le concept de classe

IV. La notion de package

V. L'encapsulation

VI. Les associations

VII. L'héritage

VIII. Le polymorphisme

IX. Les classes abstraites et les interfaces

X. Les classes internes

IV . La notion de package

Un **package** en Java est un mécanisme pour **organiser les classes** en espaces de noms.

Les packages peuvent être stockés dans des fichiers compressés appelés **archives JAR**.

Les programmeurs utilisent également les packages pour organiser des classes appartenant à une même catégorie ou fournissant des fonctionnalités similaires.

```
package moyensTransports;  
public class Velo  
{  
    String marque;  
  
    public void accelerer()  
    {  
        vitesse = vitesse + 10;  
    }  
}
```

IV . La notion de package

Package
moyensTransports

```
public class Velo
{
    String marque;
    int vitesse;
    String couleur;
    int poids;

    public void accelerer()
    {
        vitesse = vitesse + 10;
    }
    public void freiner()
    {
        vitesse = vitesse - 10;
    }
}
```

classe Velo

```
public class Voiture
{
    String marque;
    int vitesse;
    String couleur;
    int poids;

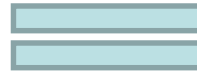
    public void accelerer()
    {
        vitesse = vitesse + 100;
    }
    public void freiner()
    {
        vitesse = vitesse - 100;
    }
}
```

classe Voiture

IV . La notion de package

Si une classe d'un programme doit utiliser une ou plusieurs classes d'un package quelconque, celle-ci doit importer le package grâce au mot-clé **import** placé en début du fichier de la classe:

```
import moyenTransport.Velo;  
import moyenTransport.Voiture;
```



```
import moyenTransport.*;
```

On peut également utiliser la notation suivante (déconseillée):

```
moyenTransport.Velo monVelo = new moyenTransport.Velo();
```


Aperçu du chapitre

I. POO <> Procédural

II. Penser le monde en objets

III. Le concept de classe

IV. La notion de package

V. L'encapsulation

VI. Les associations

VII. L'héritage

VIII. Le polymorphisme

IX. Les classes abstraites et les interfaces

X. Les classes internes

V . L'encapsulation

L'**encapsulation** permet de réunir au sein d'une même classe des attributs et des méthodes.

Cette technique s'accompagne également d'un **système de protection** qui permet de **contrôler l'accès** aux données et traitements de la classe.

Il est important que seule la classe concernée ne modifie la valeur de ses attributs. Pour cela, les **attributs** (variables d'instance) seront rendus **privés**.

Afin d'accéder aux valeurs des attributs, on fera appel aux services offerts par la classe. Ces services seront matérialisés par des **méthodes publiques**.

Ainsi, il est possible de contrôler la valeur que prend un attribut.

V . L'encapsulation – *Les modificateurs d'accès*

Comme nous l'avons vu plus haut les classes, attributs de classe et méthodes de classe peuvent avoir un **modificateur d'accès**.

En Java les modificateurs d'accès sont utilisés pour protéger l'accessibilité des attributs, des méthodes et des classes.

Ces accès sont contrôlés comme dans le tableau suivant:

Modificateur d'accès	Classe	Package	Sous-Classe	Reste du programme
<i>aucun</i>	Y	Y	N	N
public	Y	Y	Y	Y
protected	Y	Y	Y	N
private	Y	N	N	N

V . L'encapsulation – *Les getters et les setters*

Les données déclarées private à l'intérieur d'une classe ne sont accessibles et ne peuvent être modifiées que par des méthodes définies dans la même classe.

Un **accesseur** est une méthode publique qui donne l'accès à une variable d'instance privée.

Par convention, les accesseurs en lecture ou **getters** commencent par « **get** » (sauf pour les attributs boolean qui commencent par « **is** ») et les accesseurs en écriture ou **setters** commencent par « **set** ».

Exemple :

```
public int getNumero() {  
    return numero;  
}
```

```
public void setNumero(int num) {  
    numero = num;  
}
```

V . L'encapsulation – Exercices

1. Construisez une classe Point permettant de manipuler un point du plan de coordonnées de type double. Cette classe comportera :

- Un constructeur recevant en arguments les coordonnées d'un point
- Un constructeur créant un point à l'origine (0,0)
- Une méthode retournant l'abscisse du point
- Une méthode retournant l'ordonnée du point
- Une méthode retournant l'état du point (appeler cette méthode toString())
- Une méthode effectuant une translation définie par ses deux arguments de type double
- Une méthode calculant la distance entre 2 points $[(x1-x2)^2 + (y1-y2)^2]^{1/2}$
- Un attribut qui indique le nombre total de points
- Un attribut abscisse et un attribut ordonnée
- Une méthode qui retourne le nombre de points
- Une méthode qui effectue une symétrie orthogonale d'axe X
- Une méthode qui effectue une symétrie orthogonale d'axe Y
- Une méthode qui effectue une symétrie centrale

V . L'encapsulation – Exercices

2. Un produit est caractérisé par une référence et un libellé. Il est possible d'augmenter sa quantité en stock suite à un réapprovisionnement et de diminuer celle-ci suite à une demande. La TVA d'un produit est variable et est utilisée pour calculer le prix à payer suivant la quantité achetée. Attention, si la quantité achetée dépasse 100 unités, une réduction de 5% est accordée.

Utilisez deux constructeurs : un qui garnit chaque attribut et un qui impose un taux de TVA égal à 21%.

V . L'encapsulation – Exercices

3. Soit une classe Placement. Définissez une méthode capable de calculer la valeur final du capital selon sa durée (en année).

Formule à utiliser :

capital initial * (1 + taux d'intérêts composés / 100)^{nombre d'années}

De plus, créez trois constructeurs.

V . L'encapsulation – Exercices

4. Construisez une classe PorteMonnaie permettant de gérer le contenu d'un porte-monnaie. Un porte-monnaie contient des pièces de 1, 2, 5, 10, 20 et 50 centimes et de 1 et 2€. Il faut que le porte-monnaie contienne le moins de pièces possible. Cette classe comportera les méthodes suivantes :

- Une méthode « montant » qui renvoie le montant total contenu dans le porte-monnaie
- Une méthode « ajouterArgent » permettant d'ajouter de l'argent dans le porte-monnaie (n'importe quel montant)
- Une méthode « retirerArgent » permettant de retirer de l'argent (n'importe quel montant, mais pas plus que le contenu du porte-monnaie)
- Une méthode « vider » permettant de vider entièrement le porte-monnaie
- Une méthode « repartition » qui calcule le nombre de pièces de chaque type
- Une méthode « toString » permettant d'afficher le détail du contenu du porte-monnaie

Dans un premier temps, utilisez 1 attribut par pièce de monnaie.

Dans un deuxième temps, remplacer les attributs par un tableau.

Exemple : 3,88€ ➔ $1 * 2\text{€} + 1 * 1\text{€} + 1 * 50 \text{ cent} + 1 * 20 \text{ cent} + 1 * 10 \text{ cent} + 1 * 5 \text{ cent} + 1 * 2 \text{ cent} + 1 * 1 \text{ cent}$

V . L'encapsulation – Exercices

5. Créer la classe Jour. Celle-ci est composée d'un attribut représentant le numéro du jour dans la semaine (0 = dimanche, 6 = samedi), ainsi qu'une constante qui contient les deux premières lettres de chaque jour de la façon suivante:

« DILUMAMEJEVESA ». La classe contiendra 2 constructeurs :

- un qui ne reçoit pas d'argument (jour = dimanche)
- un qui reçoit comme argument une chaîne de caractère représentant le jour (chaîne à convertir en entier).

La classe contiendra les méthodes suivantes :

- toString qui retourne l'intitulé complet du jour de la semaine
- une méthode qui permet d'avancer d'un jour dans la semaine
- une méthode qui permet de renvoyer un objet Jour correspondant au jour précédent celui de l'objet sur lequel la méthode est appelée

Aperçu du chapitre

I. POO <> Procédural

II. Penser le monde en objets

III. Le concept de classe

IV. La notion de package

V. L'encapsulation

VI. Les associations

VII. L'héritage

VIII. Le polymorphisme

IX. Les classes abstraites et les interfaces

X. Les classes internes

VI . Les associations

Une **association** est une **relation** forte et permanente entre **deux classes**. Autrement dit, cette association pourra être utilisée dans n'importe quelle méthode de la classe où l'association est matérialisée.

Une association va se **matérialiser** par un **attribut** représentant l'**objet** en relation.

```
public class Cycliste
{
    String nom;
    String prenom;
    int age;
    Velo velo;
    ...
}
```

VI . Les associations – *La composition*

Une **composition** est une association plus forte pour laquelle on peut dire qu'un objet est composé d'autres objets ou qu'un objet fait partie d'un autre objet.

L'objet qui en **contient** un autre est appelé objet **composite**.

L'objet qui en **fait partie** d'un autre est appelé objet **composant**.

L'objet composite est responsable de la création de l'objet composant. Donc, si on supprime le composite, on supprime le composant.

VI . Les associations – *La composition*

```
public class Club {  
    private String nom;  
    private String adresse;  
    private String activite;  
    private Membre membres[];  
    private int nb;  
  
    public Club (String nom, String adresse, String activite, int nbMembres) {  
        this.nom = nom;  
        this.adresse = adresse;  
        activite = activite;  
        membres = new Membre[nbMembres];  
        nb = 0;  
    }  
  
    public ajouterMembre (String nom, String prenom) {  
        membres[nb] = new Membre(nom, prenom);  
        nb++;  
    }  
}
```

VI . Les associations – *L'agrégation*

L'**agrégation** est aussi appelée **composition faible**.

Dans ce cas, l'objet composite n'est plus responsable de la création de l'objet composant. Donc, si on supprime le composite, le composant n'est pas supprimé.

L'objet composant peut également se retrouver dans plusieurs objets composites, à l'inverse de la composition.

VI . Les associations – L'agrégation

```
public class Zoo {  
    private String nom;  
    private String adresse;  
    private Animal animaux[];  
    private int nb;  
  
    public Club (String nom, String adresse, int nbAnimaux) {  
        this.nom = nom;  
        this.adresse = adresse;  
        this.animaux = new [nbAnimaux];  
        nb = 0;  
    }  
  
    public ajouterAnimal (Animal animal) {  
        animaux[nb] = animal;  
        nb++;  
    }  
}
```

VI . Les associations – *La dépendance*

La **dépendance** est une association **non persistante**.

Elle ne dure que pendant l'exécution de la méthode dans laquelle elle est utilisée.

La solution est de **passer l'objet** avec lequel on veut établir la dépendance dans les **arguments de la méthode**.

On n'aura donc **plus d'attribut** représentant l'objet avec lequel on établit la relation.


VI . Les associations – Exercices

1. On dispose d'une classe Point permettant de manipuler les points d'un plan.

Réalisez une classe Segment disposant des méthodes suivantes :

- `public Segment(Point or, Point ext)`
- `public Segment(double xOr, double yOr, double xExt, double yExt)`
- `public double calculLongueur()`
- `public void deplaceOrigine(double dx, double dy)`
- `public void deplaceExtremite(double dx, double dy)`
- `public Segment symetrieCentrale()`

Réalisez une classe Triangle disposant des méthodes suivantes :

- `public Triangle(Point a, Point b, Point c)`
- `public Triangle(double x1, double y1, double x2, double y2, double x3, double y3)`
- `public Segment[] cotes()`
- `public double calculPerimetre()`
- `public double calculAire()`
- `public boolean isRectangle()`
-  `public Triangle symetrieCentrale()`

VI . Les associations – Exercices

2. Une Tache est composée d'un nom et d'un montant.

Une Facture est composée d'un numéro de facture (String), d'un numéro de TVA (String), d'un taux de TVA, d'un ensemble de 4 Taches maximum et d'une ristourne.

Il faut :

- Prévoir deux constructeurs
- Pouvoir récupérer le nombre de Factures existantes
- Ajouter un montant défini à la ristourne accordée
- Obtenir le montant de la facture (ne pas oublier de prendre en compte la ristourne et le taux de TVA)
- Pouvoir récupérer une Tache à partir de son indice
- Pouvoir ajouter une Tache (éviter les doublons)
- Pouvoir récupérer le nombre de Taches
- Pouvoir supprimer une Tache en envoyant celle-ci en paramètre (attention, de ne pas oublier de décaler les Taches qui suivent)
- Prévoir les méthodes equals, clone et toString

Aperçu du chapitre

I. POO <> Procédural

II. Penser le monde en objets

III. Le concept de classe

IV. La notion de package

V. L'encapsulation

VI. Les associations

VII. L'héritage

VIII. Le polymorphisme

IX. Les classes abstraites et les interfaces

X. Les classes internes

VII . L'héritage

L'**héritage** permet de bénéficier du patrimoine existant, c'est-à-dire de **réutiliser** les classes existantes.

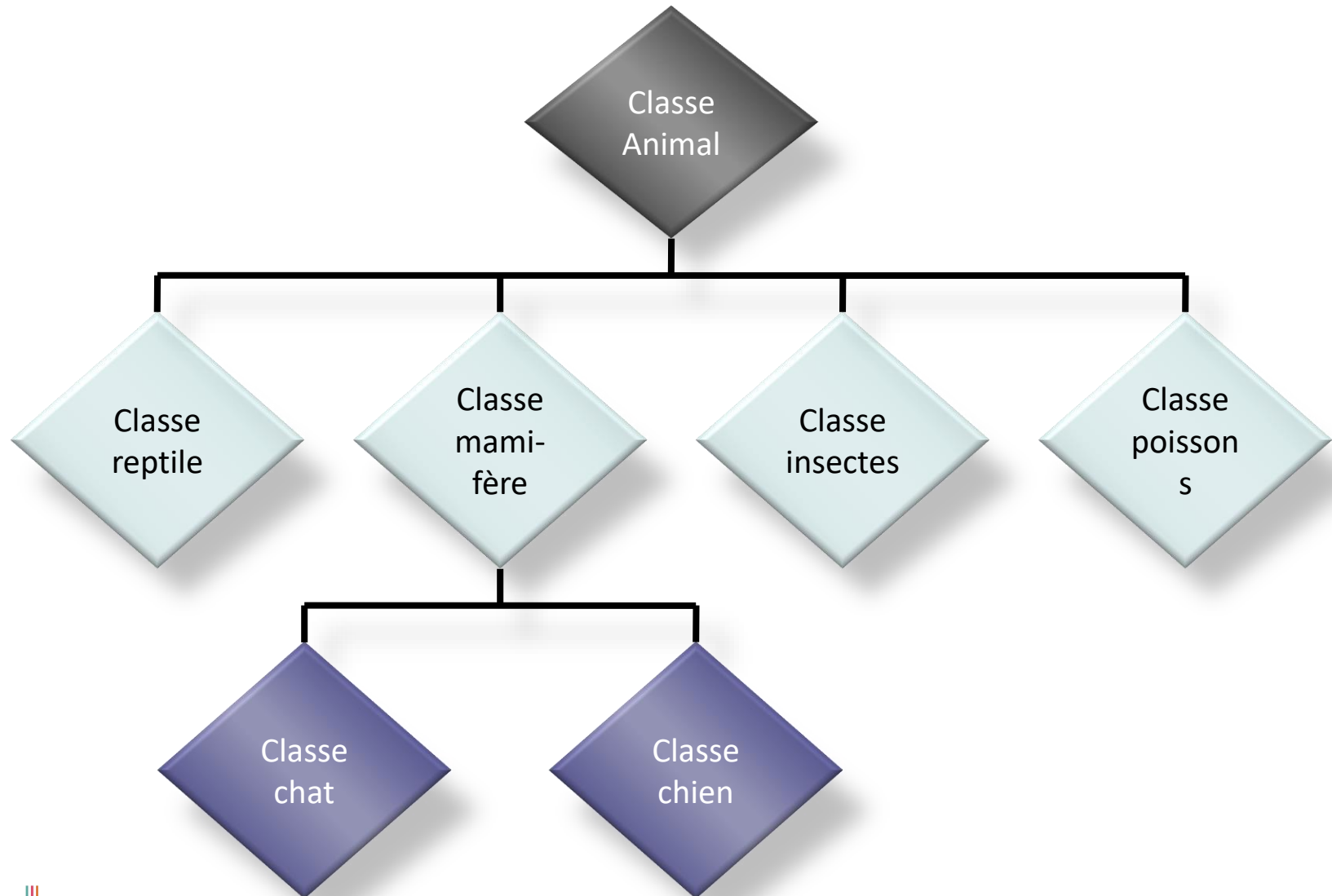
Il est possible de définir une nouvelle classe à partir d'une autre.
Ce qui signifie que la classe descendante (classe enfant ou **sous-classe**) pourra bénéficier des attributs et des méthodes de la classe de base (classe parent ou **super-classe**) dont elle hérite.

Les classes enfants peuvent contenir d'autres éléments permettant de **spécialiser** la classe parent.

Grâce à cette technique, il est donc possible de définir des **arborescences de classes** qui regroupent des classes de plus en plus spécialisées.

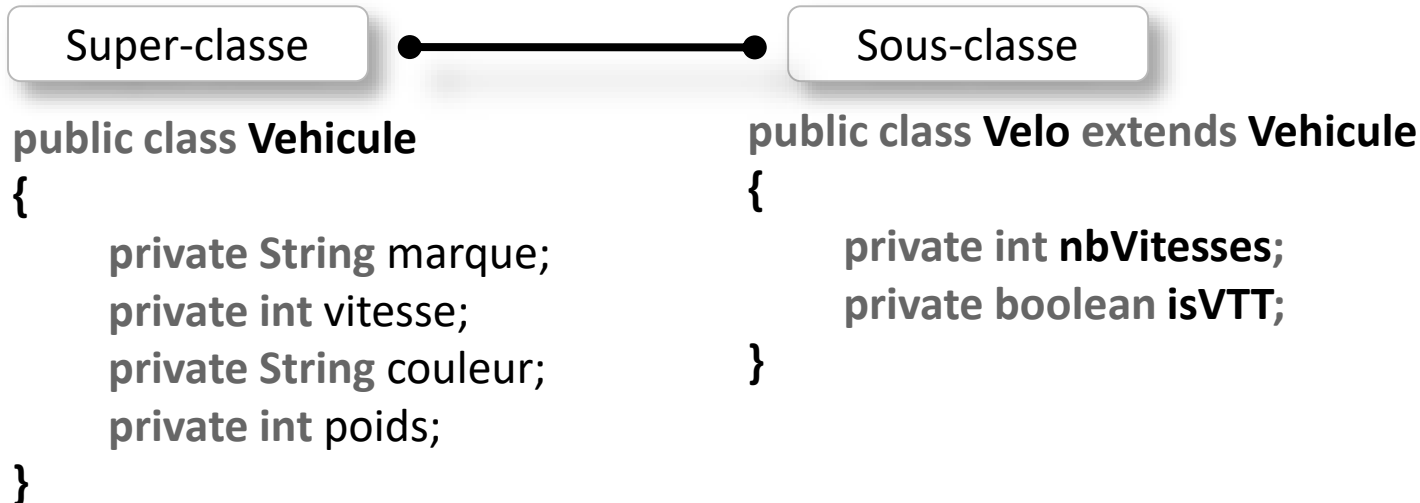
Attention, contrairement à certains langages, l'**héritage multiple** est **interdit** !

VII . L'héritage



VII . L'héritage

En Java, l'héritage est mis en œuvre au moyen du mot-clé **extends**



En l'absence d'héritage indiqué explicitement, une classe **hérite implicitement** de la classe **Object**.

Cette classe Object est la racine de la hiérarchie de classe.

VII . L'héritage – *Le constructeur*

La première instruction d'un constructeur est toujours un **appel au constructeur de la super-classe**.

Cet appel peut être **implicite**. Dans ce cas, un appel au **constructeur par défaut** est effectué.

Il est possible de spécifier à quel constructeur on souhaite faire appel grâce à un appel **explicite** à celui-ci. Ceci se fait à l'aide du mot-clé **super**.

Exemple :

```
public Velo(String marque, int vitesse, String couleur, int poids, int nbVitesses,
boolean isVTT) {
    super(marque, vitesse, couleur, poids);
    this.nbVitesses = nbVitesses;
    this.isVTT = isVTT;
}
```

VII . L'héritage – *Cast*

Le **principe de substitution** consiste à placer un objet de la sous-classe dans un référent de la super-classe.

Ceci est possible grâce au fait qu'un objet de la sous-classe peut, au moins, faire ce que fait un objet de la super-classe.

Exemple:

```
Vehicule v = new Velo();
```

Faire passer un objet de la super-classe pour un objet de la sous-classe est du **casting explicite**. Cette démarche n'est acceptée par le compilateur que si on place un cast.

Exemple :

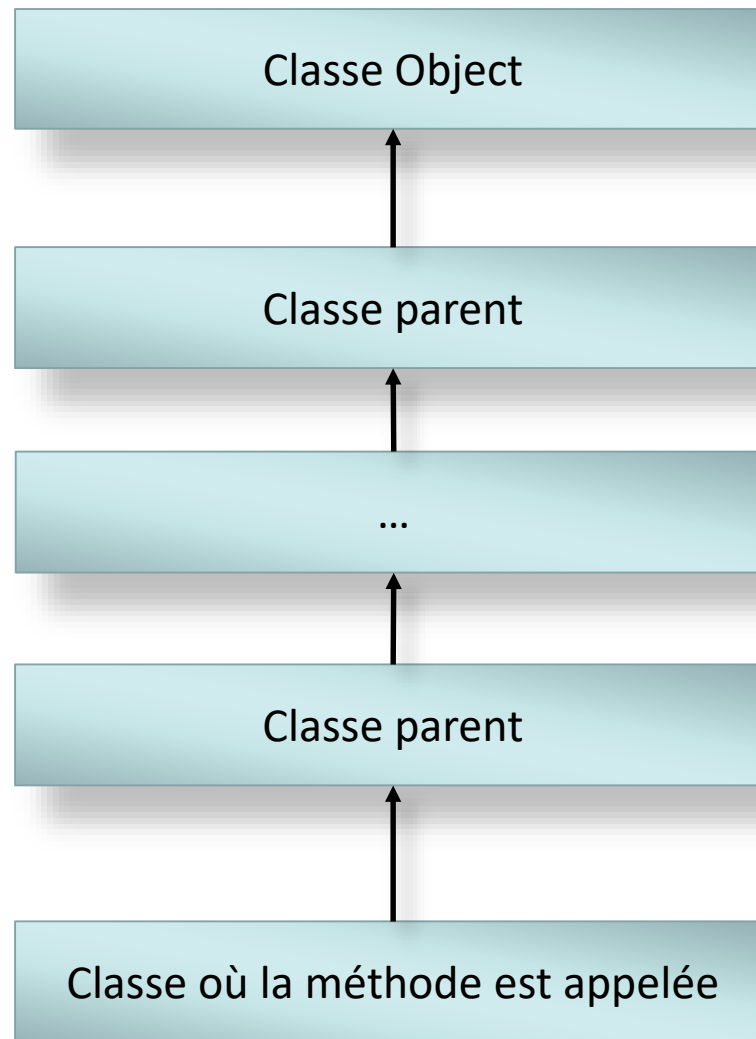
```
Velo v = (Velo) new Vehicule();
```


VII . L'héritage – *Les méthodes*

Etapas de recherche des méthodes dans la hiérarchie :

1. Lors de l'appel d'une méthode sur un objet, le compilateur va chercher celle-ci dans la **classe correspondante** au type de l'objet.
2. Si la méthode n'est pas trouvée, il continue la recherche dans la **super-classe** et ainsi de suite.
3. Si la méthode n'a pas été trouvée dans la classe Object, une erreur de compilation est lancée.

VII . L'héritage – *Les méthodes*



VII . L'héritage – *Les méthodes*

On parle de **redéfinition de méthode** lorsqu'une méthode de la sous-classe possède la **même signature** et le **même type de retour** (type dérivé autorisé) qu'une méthode de sa super-classe.

La méthode redéfinie cache la méthode de la super-classe. La sous-classe donne donc un **nouveau comportement** à cette méthode.

Lors d'une redéfinition de méthode, il est possible, dans la sous-classe, de faire appel à la méthode de la super-classe à l'aide du mot-clé **super**.

Exemple :

```
public String toString() {  
    return super.toString() + « champs spécifique sous-classe »;  
}
```

VII . L'héritage – *Les méthodes*

Les méthodes utilitaires comme **toString** et **equals** proviennent de la classe **Object**.

Ancienne définition d'`equals` (méthode propre à la classe `Personne`) :

```
public boolean equals(Personne p) {  
    // retourne la comparaison des champs de p et de this  
}
```

Nouvelle définition d'`equals` (redéfinition de la méthode de la classe `Object`) :

```
public boolean equals(Object o) {  
    if (o instanceof Personne) {  
        Personne p = (Personne) o;  
        // retourne la comparaison des champs de p et de this  
    }  
    return false;  
}
```

VII . L'héritage – *Le mot clé Final*

Le mot clé **final** ajouté devant un attribut le rend **immuable**, dès lors qu'il est initialisé.

Pour les **types primitifs**, final fige la **valeur**.

Pour les **objets**, il fige la **référence**, et non la valeur de la référence (i.e. seule l'instanciation est figée).

Devant une **méthode**, il indique que cette méthode ne peut **pas être modifiée dans une classe dérivée**.

Les méthodes static et private sont implicitement final.

Devant une **classe**, il indique que cette classe ne peut **pas avoir de sous-classe**.

Aperçu du chapitre

I. POO <> Procédural

II. Penser le monde en objets

III. Le concept de classe

IV. La notion de package

V. L'encapsulation

VI. Les associations

VII. L'héritage

VIII. Le polymorphisme

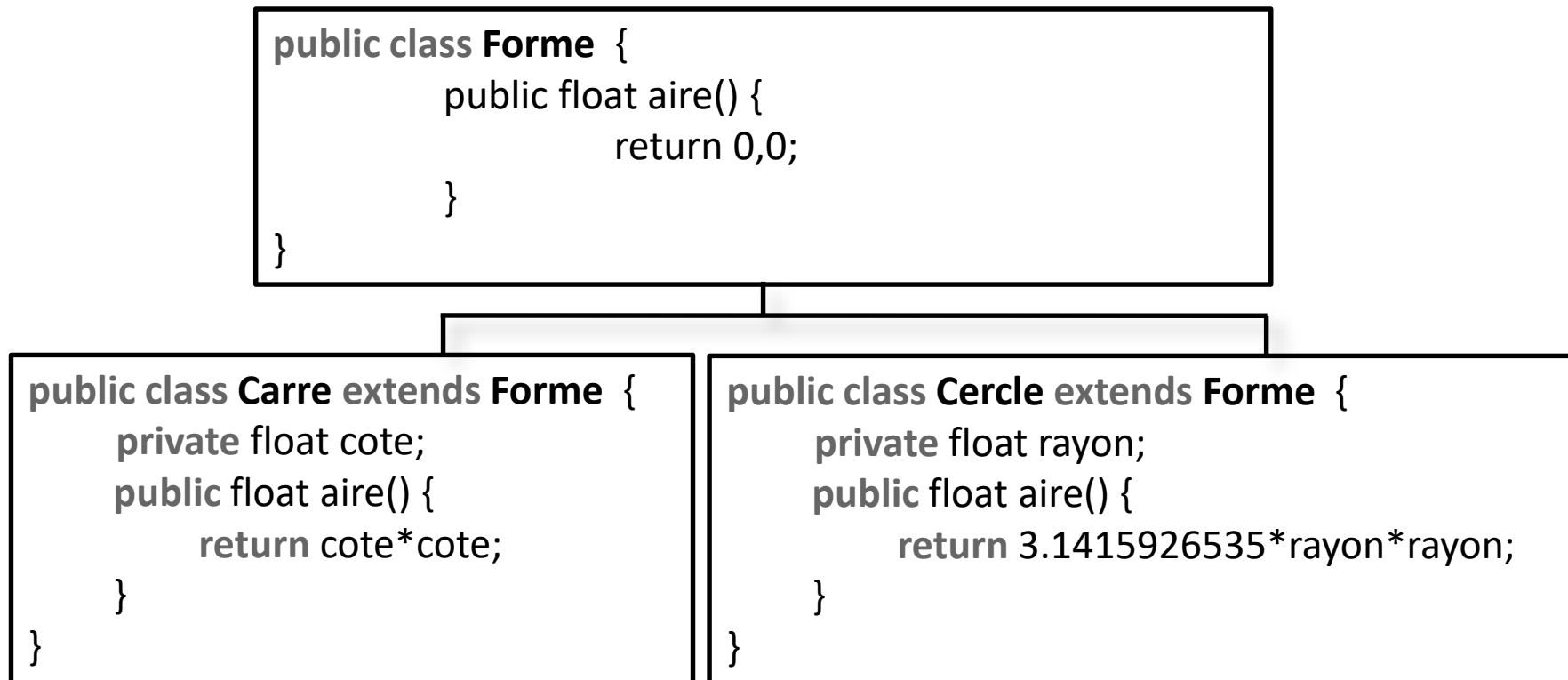
IX. Les classes abstraites et les interfaces

X. Les classes internes

VIII . Le polymorphisme

Le **polymorphisme** est une conséquence directe de l'héritage et de la redéfinition de méthode.

Il permet à une **même méthode**, dont l'existence est prévue dans la super-classe, de **s'exécuter différemment** selon que l'objet qui le reçoit est d'une sous-classe ou d'une autre.



VIII . Le polymorphisme

Exemple d'utilisation du polymorphisme:

```
Forme []tab = new Forme[5];
```

```
tab[0] = new Cercle(3);
```

```
tab[1] = new Cercle(5);
```

```
tab[2] = new Carre(4);
```

```
tab[3] = new Cercle(5);
```

```
tab[4] = new Carre(8);
```

```
for (int i = 0 ; i < tab.length ; i++) {
```

```
    // Détection de la vraie nature de tab[i] (Carre ou Cercle)
```

```
    System.out.println("Aire du " + tab[i].getClass().getSimpleName() + " : " +
```

```
    tab[i].aire());
```

```
}
```


Aperçu du chapitre

I. POO <> Procédural

II. Penser le monde en objets

III. Le concept de classe

IV. La notion de package

V. L'encapsulation

VI. Les associations

VII. L'héritage

VIII. Le polymorphisme

IX. Les classes abstraites et les interfaces

X. Les classes internes

IX . Les classes abstraites et les interfaces – *Les interfaces*

Une **interface** est une **liste de méthodes** dont on donne seulement la **signature**.
Toutes les **méthodes** d'une interface sont implicitement **publiques** et **abstraites**.

Propriétés :

- Elle peut être **implémentée par une ou plusieurs classes** qui doivent donner une implémentation pour chacune des méthodes annoncées (et éventuellement d'autres).
- Une classe peut **implémenter plusieurs interfaces**.
- Une interface peut être étendue par une ou plusieurs autre(s) interface(s).
- Une interface n'a **pas de constructeur**.
- Une interface ne peut avoir de **champs** sauf si ceux-ci sont **statiques** et **final** (constantes).

Une interface est, en réalité, une classe abstraite dont toutes les méthodes sont abstraites et dont tous les attributs sont constants.

Le mot-clé **interface** remplace le mot-clé **class** lors de la déclaration.

IX . Les classes abstraites et les interfaces – *Les interfaces*

Interface Vehicule

```
public interface Vehicule
{
    String TYPE = "Véhicules
roulant";

    void accélérer( );
    void freiner();
}
```

On a défini ici ce qu'on attend d'un objet de type véhicule

On peut maintenant donner une ou plusieurs implémentations de cette interface grâce à l'utilisation du mot clef **implements**

Classe Velo

```
public class Velo implements Vehicule
{
    String marque;
    int vitesse;

    public void accélérer()
    {
        vitesse = vitesse + 10;
    }

    public int freiner()
    {
        vitesse = vitesse - 10;
    }
}
```

Classe Voiture

```
public class Voiture implements Vehicule
{
    String marque;
    int vitesse;

    public void accélérer()
    {
        vitesse = vitesse + 50;
    }

    public int freiner()
    {
        vitesse = vitesse - 50;
    }
}
```

IX . Les classes abstraites et les interfaces – *Les interfaces*

Dans l'exemple précédent, nous avons donné deux implémentations de Vehicule.

Conséquences :

- Ces 2 objets peuvent être vus comme des véhicules, c'est ce qu'on appelle le polymorphisme.
- Partout où on attend un objet de type Vehicule, on peut mettre un de ces deux objets.
- Par ce biais, on introduit une couche d'abstraction dans notre programmation ce qui la rend beaucoup plus flexible.

IX . Les classes abstraites et les interfaces – *Les interfaces*

Si, par exemple, nous avons une classe *Personne* possédant une méthode *conduire(Vehicule v)*, on peut alors écrire :

```
Personne p = new Personne();  
p.conduire(new Auto());  
p.conduire(new Velo());
```

On peut également "instancier" un *Vehicule* par le biais de ses implémentations

```
Vehicule v = new Auto();  
Vehicule t = new Velo();
```

Dans ce cas *v* et *t* sont vus comme des *Vehicule* et, par conséquent, on ne peut appeler sur ces objets que les méthodes définies dans l'interface *Vehicule*.

IX . Les classes abstraites et les interfaces – *Les classes abstraites*

Une classe **abstraite** se trouve à mi-chemin entre les interfaces et les classes.

Comme les interfaces, elles ne sont **pas instanciables**. On ne peut en instancier qu'une sous-classe concrète. Cette **sous-classe** concrète doit **définir** toutes les méthodes abstraites.

Les classes abstraites sont déclarées par le modificateur **abstract**.

Une classe abstraite peut:

- Contenir ou hériter de méthodes abstraites (des méthodes sans corps)
- Contenir des attributs
- Avoir des méthodes normales (avec corps)

Si une classe contient au moins une méthode abstraite, celle-ci sera obligatoirement abstraite.

L'utilité d'une méthode abstraite est de **prévoir un comportement commun mais dont l'implémentation varie d'une sous-classe à une autre**.

IX . Les classes abstraites et les interfaces – *Les classes abstraites*

Interface Chien

```
public interface Chien {  
    void vieillir();  
    void aboyer();  
}
```

Classe abstraite AbstractChien

```
public abstract class AbstractChien implements Chien {  
    private int age;  
    private String couleur;  
  
    public AbstractChien(int age, String couleur) {  
        this.age = age;  
        this.couleur = couleur;  
    }  
    public void vieillir() {  
        age++;  
    }  
    public abstract void aboyer();  
}
```

IX . Les classes abstraites et les interfaces – *Les classes abstraites*

On sait que la méthode vieillir sera implémentée de la même manière quelle que soit l'implémentation de Chien.

Plutôt que d'implémenter cette interface à chaque fois, on va factoriser le code dans une classe abstraite et étendre cette classe quand le besoin s'en fait sentir.

On crée donc une classe AbstractChien qui n'implémente que la méthode vieillir de notre interface, l'autre méthode étant laissée abstract.

IX . Les classes abstraites et les interfaces – Exercices

1. Reprenez la classe Point réalisée auparavant.

Réalisez une classe Cercle qui hérite de la classe Point. Un cercle possède un rayon et dispose des méthodes suivantes :

- Un constructeur recevant en arguments les coordonnées du centre du cercle et son rayon
- Un constructeur créant un point à l'origine et recevant en argument son rayon
- `deplaceCentre` pour modifier les coordonnées du centre du cercle
- `getCentre` qui fournit en résultat un objet de type Point correspondant au centre du cercle

IX . Les classes abstraites et les interfaces – Exercices

2. Créez une interface `Forme`, représentant une forme géométrique.

Elle définit les méthodes :

- `double perimetre()`
- `double aire()`

Utilisez cette interface pour implémenter les formes géométriques suivantes :

- Carre : défini par son côté
- Cercle : défini par son rayon
- Triangle : défini par ses trois côtés

IX . Les classes abstraites et les interfaces – Exercices

3. L'interface **EtatCivil** possède l'attribut suivant : `PREFIXE_NUMERO_REGISTRE` (String)

Elle possède la méthode suivante : `getNumeroRegistreNational()`

La classe **Registre** possède l'attribut suivant : un tableau de `Personne` (la taille du tableau est à passer dans le constructeur).

Elle possède les méthodes suivantes :

- `getNombrePersonnes()`
- `ajouterPersonne(Personne)`
- `toString()`

La classe **Personne** implémente l'interface `EtatCivil`.

Elle possède les attributs suivant :

- `numeroRegistreNational` (`PREFIXE_NUMERO_REGISTRE` + numéro)
- `nom`

Elle possède les méthodes suivantes :

- `toString()`
- `equals(Object)`

IX . Les classes abstraites et les interfaces – Exercices

4. La classe Compte regroupe les caractéristiques communes aux deux classes CompteEpargne et CompteCourant. A vous de les trouver.

Un CompteEpargne possède les attributs suivants : un numéro, un nom de titulaire, un solde ainsi qu'un taux d'intérêt propre à chaque compte.

Les opérations qu'il est possible d'effectuer sur ce compte sont les suivantes : déposer de l'argent, retirer de l'argent, consulter l'état et calculer les intérêts puis les ajouter au solde du compte.

Un CompteCourant possède les attributs suivants : un numéro, un nom de titulaire, un solde ainsi qu'un découvert autorisé par la banque.

Les opérations qu'il est possible d'effectuer sur ce compte sont les suivantes : déposer de l'argent, retirer de l'argent jusqu'au niveau autorisé par le découvert, consulter l'état.

De plus, pour la classe Main, créer un menu de gestion.

IX . Les classes abstraites et les interfaces – Exercices

5. Une personne est caractérisée par un nom, un prénom et une date de naissance. Chaque personne possède une adresse (elle-même caractérisée par une rue, un numéro, une boîte, une ville et un code postal) et un compte courant. Une personne peut être amenée à déménager. Pour cela, il faut être capable de récupérer l'ensemble de ses informations (y compris son âge). Une personne a la possibilité de dépenser de l'argent et de récupérer son salaire.

Un employé possède un salaire mensuel tandis qu'un ouvrier a un salaire horaire. Un employé a un certain nombre d'heures à prester par mois (différent selon le mois). Il reçoit tous les trois mois une fiche de paie comprenant les heures qu'il a réellement prestées ces trois derniers mois et le montant total gagné. Une bonus de 100€ est offert par 5h supplémentaires et un malus de 50€ est compté par heure non prestée.

Un ouvrier reçoit tous les deux mois une fiche de paie comprenant le nombre d'heures prestées par mois durant la semaine, le nombre d'heures prestées par mois durant le week-end et le montant total gagné. Les heures prestées le week-end rapportent le triple du montant normal.

Aperçu du chapitre

- I. POO <> Procédural
- II. Penser le monde en objets
- III. Le concept de classe
- IV. La notion de package
- V. L'encapsulation
- VI. Les associations
- VII. L'héritage
- VIII. Le polymorphisme
- IX. Les classes abstraites et les interfaces
- X. Les classes internes**

X . Les classes internes

Une **classe interne** est déclarée à l'intérieur d'une autre classe.
Elle peut donc accéder aux attributs et méthodes membres de la classe externe.

Il existe les classes internes :

- **statiques** qui ne peuvent accéder dès lors qu'aux attributs et méthodes statiques de la classe externe
- **non statiques** qui peuvent accéder aux attributs et méthodes statiques de la classe ainsi qu'aux membres de l'objet qui l'a créée

En fait, le compilateur crée un membre supplémentaire dans la classe interne référençant l'objet qui l'a créé.

Une telle classe interne peut-être déclarée de manière :

- **globale** dans l'objet, elle sera accessible par l'ensemble des méthodes de l'objet.
- **locale** à une méthode de l'objet. Elle sera alors accessible depuis cette seule méthode.

X . Les classes internes – *Exemple*

```
public class ClasseExterne
{
    private int compteur = 0;
    private static String nom = "Exemple";

    static class ClasseInterne
    {
        private int index = 0;
        public ClasseInterne()
        {
            System.out.println("Création d'un objet dans " + nom);
            // impossible d'accéder à compteur
        }
    }
}
```

Classe interne statique à portée globale

La compilation du fichier ClasseExterne.java produit deux fichiers compilés :

1. ClasseExterne.class (contient la classe ClasseExterne uniquement)
2. ClasseExterne\$ClasseInterne.class (contient la classe ClasseInterne)

X . Les classes internes – *Exemple*

```
public class ClasseExterne {  
    private int compteur = 0;  
  
    class ClasseInterne {  
        private int index = 0;  
        public ClasseInterne() {  
            compteur++;  
        }  
    }  
}
```

Classe interne non statique à portée globale

Classe interne non
statique à portée locale

```
    public void methodeEnglobante() {  
        final JButton bouton = new JButton("monBouton");  
        bouton.addActionListener(new ActionListener() {  
            public void actionPerformed( ActionEvent e ) {  
                System.out.println( bouton.toString() );  
            }  
        } );  
    }  
}
```

X . Les classes internes – *Classes internes et mot clé this*

Depuis la classe interne, dans le cas où plusieurs attributs ou méthodes portent le même nom dans la classe interne et la classe externe, le pointeur `this` seul désigne l'instance de la classe interne, tandis que le pointeur `this` précédé du nom de la classe externe désigne l'instance de la classe externe.

```
public class ClasseExterne {  
    private int compteur = 10;  
  
    class ClasseInterne {  
        private int compteur = 0;  
        public void count() {  
            this.compteur++; // -> 1  
            ClasseExterne.this.compteur--; // -> 9  
        }  
    }  
}
```