Programmation Orientée Objet Java

Concepts de POO

v1.1 - 24.03.2023

Plan

- Pourquoi la POO ?
- Programmation Procédurale vs Orientée Objet
- Classe vs Objet
- Héritage
- Typage et Polymorphisme
- Surcharge et Redéfinition
- Abstraction et Classe abstraite
- Interface
- Encapsulation
- Java : La portée
- Association de classes

Pourquoi la POO ?

Pourquoi la POO ? (1/2)

Les fondements de la Programmation Orientée Objet sont issus d'une réflexion sur la vie et la qualité du logiciel.

Les 5 concepts fondamentaux de la POO sont :

- La classe
- L'objet
- L'encapsulation
- L'héritage
- Le polymorphisme

In English : Objects Oriented Programming (OOP)

Pourquoi la POO ? (2/2)

La POO avait l'objectif de respecter les 5 principes suivants :

- Avoir un niveau d'abstraction important
  => **Classes abstraites et Interfaces**
- Être modulaire
  => Découpage logique du code pour une meilleure compréhension et lisibilité
- Permettre le masquage des informations
  => Protection des données : **Encapsulation**
- Permettre l'extensibilité et la réutilisabilité
  => **Héritage** et **Composition**
- Faciliter le développement et la maintenance corrective mais aussi évolutive

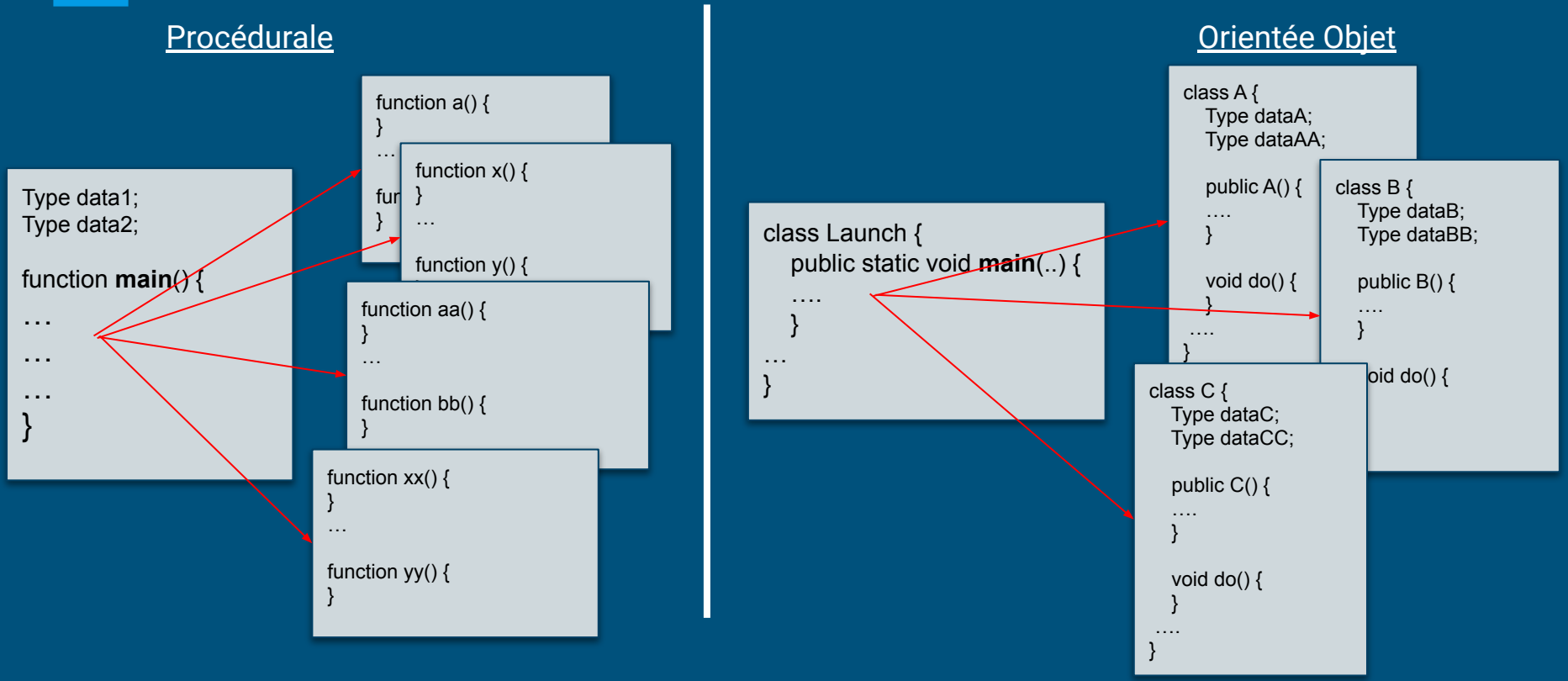
Programmation Procédurale
vs
Programmation Orientée Objet

Prog. Procédurale vs Orientée Objet

Principe de la **Programmation Procédurale** = Programmation orientée **traitements**
=> souvent des données globales
=> beaucoup de fonctions (très souvent avec des paramètres)

Principe de la **Programmation Orientée Objet** = Programmation orientée **données**
=> des données définies dans des classes
=> des méthodes dans des classes qui réalisent des traitements sur les données de la classe (pas nécessairement besoin de paramètre)
=> beaucoup de classes

Prog. Procédurale vs Orientée Objet



TD04.01-Laverie automatique

Exercice :

1. Récupérer le code Java procédural 👎 du programme de gestion de la laverie automatique
2. L'exécuter et le comprendre
3. Le réécrire en POO 👍 en respectant les consignes suivantes :
    a. Ne pas supprimer la classe initiale
    b. Créer un nouveau package pour y coder votre solution qui devra faire exactement la même chose
    c. Identifier les objets manipulés => Coder la/les classe(s) nécessaire(s)
    d. Avoir une nouvelle classe comportant une méthode "main" très simple
    e. Faire disparaître les variables "static"
    f. Et tester bien votre programme !

Classe vs Objet

Classe vs Objet

Deux notions fondamentales de la POO :

- Les **classes**
- Les **objets**

La **classe** est une représentation de quelque chose dans le monde réel.
Exemple : La classe “Voiture“ fait référence à toutes les voitures
Une **classe** est généralement définie par :

- Des attributs (propriétés)
- Des méthodes (actions)

Un **objet** est une instance de cette représentation
Exemple : Une instance “maVoitureRouge” fait référence à une voiture particulière

Classe vs Objet

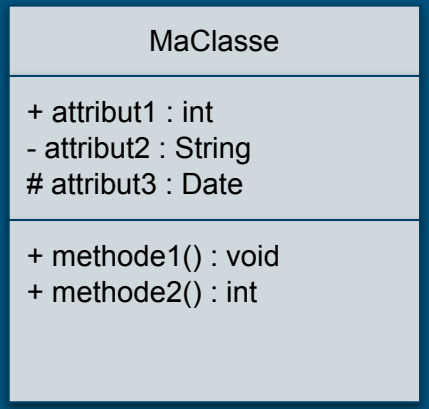
Attributs **non static** et **static**

- Un attribut (ou variable membre/d'instance) est un attribut NON static
  La valeur d'un attribut varie selon l'objet auquel il appartient
- Un attribut de classe (ou variable de classe/globale) est un attribut static

Méthodes **non static** et **static**

- Une méthode (d'instance/d'objet) est une méthode NON static
  L'exécution d'une méthode est possible par son instance (référence un objet non null)
- Une méthode de classe est une méthode static
  Elle ne peut manipuler QUE des attributs de classe c'est à dire qui sont static !!!

Classe : Diagramme de classe UML



Le + désigne la portée public
Le # désigne la portée protected
Le ~ désigne la portée “package”
Le - désigne la portée private

TD04.02-attributs et static

Exercice “Election” :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “Election”
2. Définissez un nom de package
3. Créez une classe et y ajouter la méthode public static void main(String[ ] args). Elle servira à exécuter notre programme.
4. Créez une classe Candidate contenant un attribut “nom”
5. Créez une classe Elector contenant un attribut “id” dont la valeur sera unique
    a. Ajouter la méthode public void voteFor(...) pour mémoriser le candidat du vote
6. L’objectif est de créer :
    a. Plusieurs candidats (3 max)
    b. Plusieurs électeurs (10 max)
    c. De faire voter 1 seule fois chaque électeur pour un seul candidat
7. Ouvrez le bureau de vote
    a. Création des candidats
    b. Création des électeurs
    c. Ouverture du vote
8. Affichez les résultats après dépouillement des votes
    a. Affichez pour chaque électeur, le candidat pour qui il a voté
    b. Afficher les candidats et leur nombres de votes acquis

Héritage

Héritage : Définition (1/3)

L'héritage est le mécanisme qui consiste à **définir une classe à partir d'une autre** c'est à dire qu'il permet à une classe d'hériter d'une autre.
Hériter en POO signifie bénéficier des attributs et méthodes de la classe parente.
Néanmoins, les méthodes peuvent être redéfinies au sein des sous-classes.

L'héritage consiste à définir une **classe comme étant la sous-classe d'une autre classe**.
Une sous-classe **hérite (possède) donc des attributs et méthodes** (non privés) **de sa super classe**.

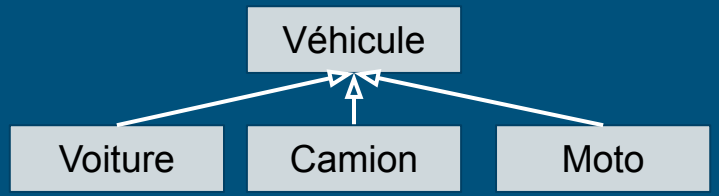
Héritage : Définition (2/3)

Une classe qui va avoir des sous-classes est appelée “super classe”, “classe parente”, “classe mère”.
Une classe qui hérite d’une classe est appelée “sous classe”, “classe fille”, “classe dérivée”.

L’héritage permet donc de définir :

- une **classe très générale** (super classe) et
- des **sous-classes plus spécialisées**.

Exemple : Si l’on définit une classe Véhicule, on pourrait définir des sous-classes de Véhicule (super classe) telles que Voiture, Camion, Moto, … qui sont tous des véhicules.



Héritage : Définition (3/3)

Une sous-classe va donc :

- être comme sa super classe
  => **bénéficie des attributs et méthodes de sa super classe**
- pouvoir avoir des méthodes supplémentaires

A retenir :

- Une instance d'une sous-classe
  - ne peut pas accéder aux attributs privés ou méthodes privées de sa super classe.
  - peut accéder aux attributs publics ou méthodes publiques de sa super classe comme s'ils étaient définis directement dans la sous-classe elle-même.
- Pour accéder au sein d'une sous-classe à un attribut ou une méthode de sa super classe, il faut préciser la déclaration avec le mot clé super.

L’héritage : Concept

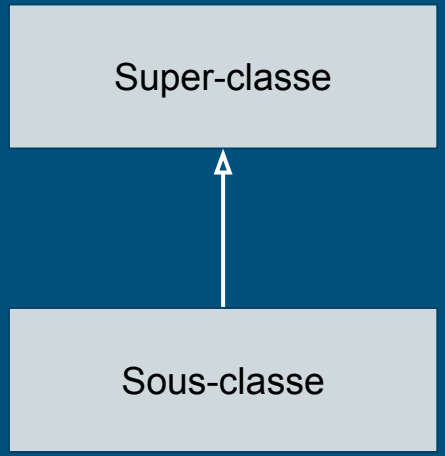
L’héritage permet de mettre en place les principes de généralisation et spécialisation.

La généralisation consiste à définir des propriétés et actions qui seront communes à plusieurs objets.
On définit généralement cela dans la **super classe**.

La spécialisation à l’inverse consiste à spécifier un objet à partir d’un objet plus général.
On définit généralement cela dans les **sous-classes**.

Exemple : On pourrait généraliser les véhicules dans un type (une classe) “Véhicule” et définir des types de véhicules particuliers tels que les autos, les motos, les camions, … qui sont des véhicules plus spécifiques.

Héritage : Diagramme de classe UML



L’héritage se représente par une flèche à la pointe creuse.

La classe “Sous-classe” **hérite** de la classe “Super-classe”.

Héritage en Java (1/3)

Une classe en Java hérite toujours d'une autre classe.
Si aucune super classe n'est précisée dans la déclaration de la classe alors elle hérite automatiquement de la classe Object.

Le mot clé en Java pour définir l'héritage est extends.

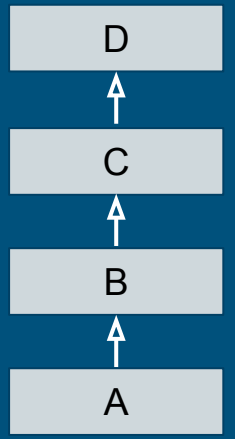
```
package fr.esgi.poo.inheritance;

public class B extends A {

}
```

La classe "B" est une sous-classe de la classe "A".
La classe "A" est la super classe de la classe "B".

Héritage en Java (2/3)

En Java, une sous-classe (classe fille) hérite/dérive d'une **unique** super classe (classe mère).
L'**héritage multiple** n'est pas supporté par le langage Java.

MAIS une classe A peut parfaitement hériter d'une classe B qui hérite d'une classe C qui hérite d'une classe D, etc.



Héritage en Java (3/3)

En Java, une classe déclarée final ne peut pas être héritée.

```
public final class A {
}
```

```
public class B extends A { // IMPOSSIBLE
}
```

Héritage en Java : Constructeur (1/2)

Si une super classe n'a pas de constructeur explicitement défini, une sous-classe de cette super classe **pourra ou non** définir un ou plusieurs constructeurs.
Si aucun constructeur n'est défini, c'est le constructeur par défaut de la classe (constructeur sans paramètre) qui est appelé.

```
public class A {
        //  Pas de constructeur explicitement défini
}
```

```
public class B extends A {
    //  Pas besoin de définir un constructeur
}
```

Héritage en Java : Constructeur (2/2)

Si une super classe a un ou plusieurs constructeurs explicitement définis, une sous-classe de cette super classe devra **obligatoirement définir aussi au moins un constructeur** dans sa classe. Sinon, une **erreur de compilation** se produira !
Par ailleurs, la première instruction du constructeur d'une sous-classe doit être l'appel à un des constructeurs de la super classe avec le mot clé super.

```
public class A {
    public A(int n) {
    }
    public A(float m) {
    }
}
```

```
public class B extends A {
    //  Pas de constructeur de défini => ERREUR COMPILATION
}
```

```
public class B extends A {
    public B(double doubleNumber) {
        super((int)doubleNumber); //  Instruction OBLIGATOIRE !!!!
    }
}
```

Héritage et instanciation

Lorsqu'on manipule un ensemble de classes avec des relations d'héritage, il est possible d'écrire :
<Type de sa SuperClasse> maClasse = new <Type de la classe>();

```
public class MaClasseParente {

}
```

```
public class MaClasseFille extends MaClasseParente {

}
```

```
MaClasseParente parente1 = new MaClasseParente(); // OK
MaClasseFille fille1 = new MaClasseFille(); // OK

MaClasseParente fille2 = new MaClasseFille(); // OK

~~MaClasseFille parente2 = new MaClasseParente(); // KO~~
```

TD04.03a-Héritage "Sportifs"

Exercice :

1. Représentez sous forme d'un schéma les entités suivantes :
   a. Avant-Centre
   b. Skieur
   c. Sportif
   d. Humain
   e. Descendeur
   f. Footballeur
   g. Gardien
   h. Slalomeur
2. Dans votre IDE, créez un nouveau projet Java nommé par exemple "Sportifs"
   a. Définissez un nom de package
   b. Créez les différentes classes correspondants avec leur relation
   c. Rajoutez le mot clé final aux "bonnes" classes

TD04.03b-Héritage "Scolarité"

Exercice :

1. Représentez sous forme d'un schéma les entités suivantes :
   a. Ecole
   b. CM2
   c. Ecole primaire
   d. Collège
   e. Ecole maternelle
   f. Lycée
   g. Petite section
   h. Terminale
   i. 4ème
   j. …
2. Dans votre IDE, créez un nouveau projet Java nommé par exemple "Scolarité"
   a. Définissez un nom de package
   b. Créez les différentes classes correspondants avec leur relation
   c. Rajoutez le mot clé final aux "bonnes" classes

La super classe Object

En Java, toute classe définie hérite de la classe Object.
Par conséquent, une variable de type Object peut être utilisée pour référencer un objet de type quelconque.

De plus, n'importe quel objet en Java possède les méthodes de Object :

- clone()
- equals()
- finalize()
- **getClass()**
- hashCode()
- notify() / notifyAll()
- **toString()**
- wait() / wait(long) / wait(long, int)

Héritage en Java : mots clés super et this

Une instance d'une classe peut faire référence :

- à sa classe parente avec le mot clé super
- à elle-même avec le mot clé this

On peut commencer le code d'un constructeur par un appel d'un constructeur de la classe de base :

- super(…),
- ou d'un autre constructeur de la classe this(…).

```
public final class A {
    public A(int n) {
    }

    public A(int x, int y) {
        this(x);
    }
}
```

Héritage en Java : mots clés super et this

Si, dans une classe, un constructeur est défini sans commencer par un appel de constructeur, Java insère un appel du constructeur par défaut de la super classe.

```
public class A {
    public A() {
    }
    public A(int n) {
    }
}
```

```
public class B extends A {
    public B(int n) {
//  On doit :
//          - soit coder super(n);
//          - soit rien mais dans ce cas, un super() est exécuté automatiquement
mais il faut pour cela que le constructeur A() ait été défini !!
    }
}
```

TD04.04-Héritage “Formes”

Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “Formes”
2. Définissez un nom de package
3. Créez une classe et y ajouter la méthode public static void main(String[ ] args) mais laissez-la vide
4. Créez une autre classe nommée Rectangle qui aura :
   a. Deux attributs entiers “largeur” et “longueur”
   b. Un constructeur pour initialiser les 2 attributs
   c. Une méthode area() pour calculer la surface du rectangle
   d. Une méthode perimeter() pour calculer le périmètre du rectangle
5. Créez une autre classe nommée Square qui aura :
   a. Un attribut entier “cote”
   b. Un constructeur pour initialiser l’attribut
   c. Une méthode area() pour calculer la surface du carré
   d. Une méthode perimeter() pour calculer le périmètre du carré
6. Implémentez dans la méthode main, l’instanciation d’un rectangle, le calcul de la surface et du périmètre. Idem pour le carré.
7. Modifier la classe Square pour en faire une sous-classe de la classe Rectangle
8. Exécutez le projet pour afficher la surface et le périmètre du rectangle et du carré

Héritage en Java

L'opérateur booléen instanceof renvoie true lorsque qu'un objet peut être converti en type défini par une classe.
Il permet donc de savoir à quelle classe appartient une instance.

Syntaxe : <référence d'une classe> instanceof <nom d'une classe>

```
public class MaClasseParente {

}
```

```
public class MaClasseFille extends MaClasseParente {

}
```

```
MaClasseParente parente = new MaClasseParente();
MaClasseFille fille = new MaClasseFille();

boolean vrai, faux;
vrai = parente instanceof MaClasseParente; // True
faux = parente instanceof MaClasseFille; // False
vrai = fille instanceof MaClasseFille; // True
vrai = fille instanceof MaClasseParente; // True !!
```

Héritage en Java

La méthode getClass() (définie dans la classe Object) permet de connaître la classe exacte d'une instance

Syntaxe : <référence d'une classe>.getClass()

```
public class MaClasseParente {

}
```

```
public class MaClasseFille extends MaClasseParente {

}
```

```
MaClasseParente maClasseParente = new MaClasseParente();
name = maClasseParente.getClass().getSimpleName(); // MaClasseParente

MaClasseFille maClasseFille = new MaClasseFille();
name = maClasseFille.getClass().getSimpleName(); // MaClasseFille

MaClasseParente maClasse = new MaClasseFille();
name = maClasse.getClass().getSimpleName(); // MaClasseFille
```

TD04.05-Héritage et instanceof

Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “SimpleHéritage”
2. Définissez un nom de package
3. Créez une classe et y ajouter la méthode public static void main(String[ ] args) mais laissez-la vide
4. Créez 3 classes nommées GrandFather, Father, Son qui sont liées par héritage. A vous de les définir correctement.
5. Ajoutez une méthode foo() dans chacune des classes qui renvoie le nombre de lettres du nom de la classe
6. Dans la méthode main()
   a. Instanciez les 3 classes
   b. Affichez le résultat de l’appel aux 3 méthodes foo()
   c. Vérifiez les types des 3 instances à l’aide du mot clé instanceof

Typage et polymorphisme

Le polymorphisme

Le polymorphisme permet de manipuler des objets sans connaître tout à fait leur type.
C'est un principe extrêmement puissant en **Programmation Orientée Objet**, qui complète l'héritage.

Le polymorphisme consiste à définir des méthodes de même signature (même prototype) dans des classes différentes. Ces méthodes feront des traitements différents.

Le polymorphisme découle de la notion de redéfinition de méthodes entre une super-classe et ses sous-classes.

Le polymorphisme est le concept consistant à fournir une interface unique à des entités pouvant avoir différents types.

Le polymorphisme

Le polymorphisme permet de définir dans la super-classe des méthodes tenant compte des spécificités des sous-classes (classes dérivées).

En cas de redéfinition de méthodes, c'est la méthode de la classe dérivée de l'objet qui est prise en compte, et non celle de la super-classe.

Le choix de la méthode à appliquer se fait automatiquement en fonction du type effectif de l'objet et non pas en fonction du type de la variable qui référence l'objet.
Ce choix porte le nom de **liaison dynamique** (ou ligature dynamique).

Le polymorphisme : Intérêt

En proposant d'utiliser un même nom de méthode pour plusieurs types d'objets différents, le polymorphisme permet une **programmation beaucoup plus générique**.

Le développeur n'a pas à savoir, lorsqu'il programme une méthode, le type précis de l'objet sur lequel la méthode va s'appliquer.
Il lui suffit de savoir que cet objet implémentera la méthode.

Surcharge et Redéfinition

Méthodes : Surcharge

La **surcharge** (**overload** en anglais) est le terme qui désigne le fait de définir plusieurs fois une même méthode avec des arguments différents (nombre et/ou types différents).

On parle de surcharge lorsqu'un “même symbole” possède plusieurs significations différentes choisies en fonction du contexte d'utilisation.
Par exemple, l'opérateur “+” peut, suivant les cas, correspondre à une somme d'entiers, de flottants ou une concaténation de chaînes de caractères.

La surcharge s'applique en Java : plusieurs méthodes peuvent porter le même nom pour peu que le nombre et le type de leurs arguments permettent au compilateur d'effectuer son choix.
Le type de la valeur de retour n’a pas d’incidence dans la surcharge.

Méthodes : Surcharge

Exemple de **surcharge** de la méthode “faireQqch” :

```
public class MaClasse {

  void faireQqch() {
  }

  void faireQqch(String s) {
  }

  void faireQqch(String s, int a) {
  }

  void faireQqch(String s, boolean b) {
  }
}
```

TD04.06-Surcharge

Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “Surcharge”
2. Définissez un nom de package
3. Créez une autre classe nommée Ambiguous qui aura :
   a. Une méthode calculate(int a, long b) qui renvoie 10*a + b
   b. Une méthode calculate(long a, int b) qui renvoie 1000*a + b
4. Créez une classe et y ajoutez la méthode public static void main(String[ ] args) qui va instancier un objet Ambiguous et :
   a. Essayez d’appeler la 1ère méthode calculate avec les valeurs 1 et 2
   b. Essayez d’appeler la 2ème méthode calculate avec les valeurs 2 et 4

   Que constatez-vous ?

Méthodes : Redéfinition

La **redéfinition** (**override** en anglais) consiste à réécrire dans une sous-classe, une méthode définie dans la super-classe et à changer son comportement.

NB :
Le nombre et le type des arguments ainsi que la valeur de retour doivent être exactement les mêmes.

Méthodes : Redéfinition

Exemple de **redéfinition** de la méthode “faitCela” :

```
public class MaClasseParente {
  void faitCi() {
     System.out.println("MaClasseParente.faitCi");
  }
  void faitCela() {
     System.out.println("MaClasseParente.faitCela");
  }
}

public class MaClasseFille extends MaClasseParente {
  void faitCela() {
     System.out.println("MaClasseFille.faitCela");
   }
}
```

```
ClasseParente parente = new MaClasseParente();
MaClasseFille fille = new MaClasseFille();

parente.faitCi(); // affiche "MaClasseParente.faitCi"
parente.faitCela(); // affiche "MaClasseParente.faitCela"

fille.faitCi(); // affiche "MaClasseParente.faitCi"
fille.faitCela(); // affiche "MaClasseFille.faitCela”
```

Méthodes : Redéfinition

Remarques :

1. Ne pas confondre la **redéfinition** avec la **surcharge** :
   a. **Redéfinition** : même type de retour et mêmes paramètres
      => une méthode ne peut être redéfinie QUE dans une sous-classe
   b. **Surcharge** : type de retour et paramètres différents, les paramètres au moins doivent être différents.
2. Une classe définie avec le modificateur d'accès final ne peut pas être dérivée.
3. Une méthode définie avec le modificateur d'accès final ne peut pas être redéfinie dans les classes dérivées.
4. Les méthodes static ne sont pas concernées par la redéfinition.

TD04.07-Héritage et redéfinition

Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “Redéfinition”
2. Définissez un nom de package
3. Créez une classe et y ajouter la méthode public static void main(String[ ] args) mais laissez-la vide
4. Créez une classe nommée Sum qui aura :
   a. Deux attributs entiers
   b. Un constructeur pour initialiser les 2 attributs
   c. Une méthode compute() pour calculer l’addition des 2 attributs
5. Faites de même avec une classe Substract
   Ne pourrait-on pas alléger le code avec de l’héritage ?
6. Modifiez votre code en conséquence
7. Implémentez une classe Multiply basée sur le même principe
8. Testez (c’est prouver ;)) tout cela !

Abstraction

L’abstraction

L’abstraction consiste à ne tenir compte que de principes généraux sans tenir compte de leurs détails.

En Java, ce principe peut être implémenté à l’aide des **Classes Abstraites**.

Classe abstraite (1/4)

Une **classe abstraite** est une classe qui ne permet pas d'instancier des objets.
Elle ne peut donc servir que de classe de base afin d'être héritée.

Une classe abstraite possède au moins une méthode abstraite.
Une méthode abstraite est une méthode définie uniquement par sa signature et aucun code.

La sous-classe d'une classe abstraite devra obligatoirement redéfinir toutes les méthodes abstraites de sa super classe à moins qu'elle soit elle-même abstraite.

Classe abstraite (2/4)

Une classe abstraite, déclarée avec le mot clé abstract, est une classe qui ne peut pas être instanciée*.

Dès qu'une classe contient une méthode abstract, elle doit obligatoirement être elle-aussi déclarée abstract.

L'intérêt des classes abstraites :

- Permet de spécifier toutes les fonctionnalités que l'on souhaite disposer dans les sous-classes, c'est à dire, définir la déclaration de méthodes qui devront être obligatoirement implémentées par ses sous-classes.
- Permet d'exploiter le polymorphisme en étant sûr que certaines méthodes existent.

*A moins d'implémenter les méthodes abstraites en créant une classe anonyme

Classe abstraite (3/4)

Elle pourra avoir plusieurs de ses méthodes déclarées avec le mot clé abstract.

Une méthode abstraite :

- Est déclarée comme une méthode classique, avec ou non des paramètres en entrée et un type de sortie ou void
- **MAIS** n'a pas de corps de méthode => n'a pas de code
- Doit être **obligatoirement** déclarée dans ses sous-classes (non abstraites) avec son code

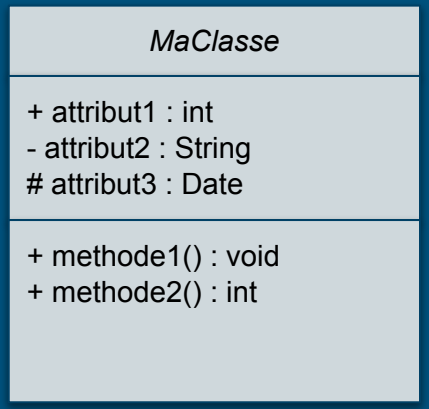
Seule une classe abstraite peut contenir des méthodes abstraites.

Classe abstraite (4/4)

```java
public abstract class MaClasseAbstraite {
  void uneMéthode() {
        // Code
  }
  abstract void uneMethodeAbstraite();
  abstract void uneAutreMethodeAbstraite();
}

public abstract class MaSousClasseAbstraite extends MaClasseAbstraite {
  @Override
  void uneMethodeAbstraite() {
        // Code
  }
}

public class MaClasseConcrete extends MaSousClasseAbstraite {
  @Override
  void uneAutreMethodeAbstraite() {
        // Code
  }
}
```

```java
MaClasseAbstraite mca = new MaClasseAbstraite(); // Erreur

MaSousClasseAbstraite msca = new MaSousClasseAbstraite(); // Erreur

MaClasseConcrete mcc = new MaClasseConcrete(); // OK
```

Classe abstraite : Diagramme de classe UML



Le + désigne la portée public
Le # désigne la portée protected
Le ~ désigne la portée “package”
Le - désigne la portée private

Remarque :
Une méthode abstract est toujours public

TD04.08-Classe abstraite

Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “Figures”
2. Définissez un nom de package
3. Créez une classe et y ajouter la méthode public static void main(String[ ] args) mais laissez-la vide
4. Créez des classes :
   a. Classe Rectangle qui aura une méthode draw()
   b. Classe Triangle qui aura une méthode draw()
   c. Classe Circle qui aura une méthode draw()
5. Dans la méthode main, créez 2 rectangles, 3 triangles et un cercle. Essayez de stocker ces 6 figures dans un tableau ?
6. Créez une classe abstraite nommée Shape et faites que les classes héritent de cette classe abstraite.
   Essayez à nouveau de stocker ces 6 figures dans un tableau ?
7. Rajoutez une méthode abstract draw() dans la classe abstraite
8. Créez une nouvelle classe Square qui hérite aussi de Shape.
   Que doit contenir cette classe ?
9. Instanciez un carré que vous stockerez dans votre tableau de figures.
10. Exécutez le projet pour afficher les méthodes draw() de vos figures.

Interface

Java : Interface (1/5)

Une **interface** Java définit un comportement qui devra être implémenté par une classe MAIS l'**interface** ne l'implémente pas (ne contient pas de code), sauf depuis **Java 8**.

Il s'agit en quelque sorte de définir un contrat.

Une **interface** va pouvoir définir :

- des méthodes par leur signature uniquement
- des constantes
- des méthodes avec du code **(Java 8+)**

```
public interface MonInterface {
  int FIELD = 10; // equivaut à public static final int FIELD = 10;

  void uneMethode(); // equivaut à public abstract void uneMethode();
}
```

Une **interface** n'implémente AUCUNE méthode (mais déclare des prototypes de méthodes), sauf depuis **Java 8**.
Une **interface** n'a pas de constructeur.

Java : Interface (2/5)

Les interfaces permettent de regrouper des classes ayant une ou plusieurs propriétés communes et de déclencher des traitements pour les objets des classes implémentant cette interface.

Une **interface** est un peu comme une classe abstraite MAIS où toutes les méthodes sont abstraites.
Une **interface** peut dériver de plusieurs **interfaces**.

Une **classe** :

- ne peut hériter que d'une seule super-classe
- MAIS peut implémenter plusieurs interfaces

Java : Interface (3/5)

Une **interface** est en général définie au sein d'un fichier spécifique qui a l'extension ".java".

Pour définir une **interface**, elle doit être préfixée par le mot réservé interface

Le nom d'une **interface** doit respecter la nomenclature PascalCase.
Exemple : interface MonInterface { … }

Remarque : Il était courant par le passé d'avoir des interfaces avec un nom préfixé d'un "i" majuscule.

Java : Interface (4/5)

Déclaration d'une **interface** :

```java
public interface MaBaseInterface2 {
    void f2();
}
```

```java
public interface MaBaseInterface1 {
    void f1();
}
```

```java
public interface MonInterface extends MaBaseInterface1, MaBaseInterface2 {
    void f();
}
```

```java
public class MaClasseAvecInterface implements MonInterface {
    @Override
    public void f1() {
        // issue de MaBaseInterface1
    }

    @Override
    public void f2() {
        // issue de MaBaseInterface2
    }

    @Override
    public void f() {
        // issue de MonInterface
    }
}
```

Pour spécifier le(s) **interface(s)** implémentée(s) par une classe, on utiliser le mot réservé implements (et non extends comme pour les classes)

Java : Interface (5/5)

La portée des interfaces est public ou “package” (sans mot clé).

Les (signatures des) méthodes déclarées au sein de l’interface sont public ou “package” (sans mot clé).

A la compilation, une interface (définie dans un fichier .java) se compile en un fichier .class comportant son nom.

Exemple : “MonInterface.java” devient après compilation “MonInterface.class”

Java : Interface

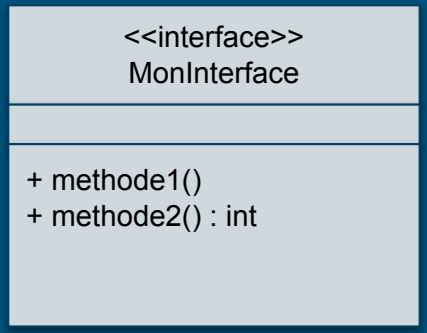
Deux nouvelles possibilités ont vu le jour dans les interfaces avec **Java 8** :

- Les méthodes default
  Ces méthodes permettent d'avoir une implémentation (du code) par défaut afin que les classes qui implémentent cette interface ne soient pas obligées de redéfinir cette méthode.
  Il suffit de les déclarer avec le mot réservé default
  Exemple : default void foo() { .... }
- Les méthodes static
  Ces méthodes permettent d'avoir une implémentation (du code) MAIS que les classes qui implémentent cette interface ne puissent redéfinir.
  Il suffit de les déclarer avec le mot réservé static
  Exemple : static boolean bar() { .... }

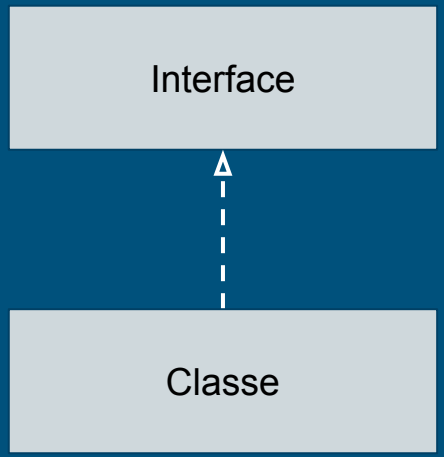
Depuis **Java 9**, on peut aussi définir des méthodes private et des méthodes static private

Interface : Diagramme de classe UML

Le + désigne la portée public
Le ~ désigne la portée “package”



Interface : Diagramme de classe UML



La réalisation d'une interface par une classe se représente par une flèche pointillée à la pointe creuse.

Java : Interfaces courantes

Quelques interfaces des API Java très courantes :

- Comparable
- Cloneable
- Iterator et ListIterator
- Runnable
- Serializable
- ...

TD04.09-Interface

Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “FormesGeom”
2. Définissez un nom de package
3. Créez une classe et y ajouter la méthode public static void main(String[ ] args) mais laissez-la vide pour le moment
4. Créez des classes :
   a. Classe Square qui aura une méthode computeArea()
   b. Classe Rectangle qui aura une méthode computeArea()
5. Dans la méthode main, créez 2 rectangles, 3 carrés. Essayez de stocker ces 5 formes géométriques dans un tableau ?
6. Créez une interface nommée Areable et faites que les classes implémentent cette interface. Essayez à nouveau de stocker ces 5 formes géométriques dans un tableau ?
7. Pouvez vous rajouter la méthode computeArea() dans l’interface Areable ?
8. Créez une nouvelle classe Circle qui implémente l’interface. Que devez-vous rajouter ?
9. Instanciez un objet que vous stockerez dans votre tableau.
10. Exécutez le projet pour afficher les méthodes computeArea() de vos formes.

Encapsulation

L’encapsulation : principe

L’encapsulation est l'idée de cacher l'information contenue dans un objet et de ne proposer que des méthodes de manipulation de cet objet.

En général, une bonne programmation objet protège les données relatives à un objet : les attributs définis dans une classe.
Pour cela, on déclare ces champs private, et on utilise des méthodes (getter/setter) déclarées public pour y accéder.

L’encapsulation et Java

Le langage Java répond au principe d’**encapsulation** grâce aux niveaux de portée/visibilité que l’on peut appliquer sur les **attributs** et sur les **méthodes** :

- public : attributs et méthodes sont accessibles à tous
- protected : attributs et méthodes sont accessibles seulement à la classe elle-même et aux classes dérivées
- “package” : attributs et méthodes sont accessibles seulement à la classe elle-même et aux classes du même package
- private : attributs et méthodes sont accessibles seulement à la classe elle-même

Java : La portée

La portée : Généralités

La **portée** (ou **visibilité** ou **accessibilité** ou **droits d'accès**) (access level en EN) permet de définir le niveau d'accès au composant auquel elle fait référence.
On parle aussi d'autorisations/modificateurs d'accès.

La portée : Pourquoi ?

- Préservation de la sécurité des données
  - Les données privées sont simplement inaccessibles de l'extérieur
  - Elles ne peuvent donc être lues ou modifiées que par les méthodes d'accès rendues publiques
- Préservation de l'intégrité des données
  - La modification directe de la valeur d'une variable privée étant impossible, seule la modification à travers des méthodes spécifiquement conçues est possible, ce qui permet de mettre en place des mécanismes de vérification et de validation des valeurs de la variable
- Cohérence des systèmes développés en équipes
  - Les développeurs de classes extérieures ne font appel qu'aux méthodes et, pour ce faire, n'ont besoin que de connaître la signature. Leur code est donc indépendant de l'implémentation des méthodes

La portée : 4 niveaux en Java

4 niveaux existent en Java dont la liste est ordonnée par restriction croissante :

- public
  Accessibilité depuis n'importe où
- protected
  Accessibilité depuis toutes les classes qui sont dans le même paquetage que A, et également depuis celles qui ne sont pas dans le même paquetage mais qui héritent de la classe A
- Sans mot clé mais dit "package" (portée par défaut)
  Portée équivalente à public mais **limitée au package**
- private
  Accessibilité uniquement depuis l'intérieur de la classe A

Ces niveaux sont **exclusifs** ! Un attribut ou une méthode ne peuvent avoir qu'un seul niveau d'accès.

La portée : Composants

Les composants pouvant avoir une portée sont :

- Les **classes** et **interfaces** : Autorisation ou non à l'instanciation et accès aux attributs et méthodes
- Les **constructeurs** : Autorisation ou non à l'instanciation et accès aux attributs et méthodes
- Les **attributs** : Autorisation ou non de la lecture/écriture
- Les **méthodes** : Autorisation ou non à l'appel

Ne peuvent avoir une déclaration de portée :

- Les paramètres des constructeurs et méthodes
- Les variables locales

La portée : Autorisations d'accès

Les classes et interfaces, constructeurs, attributs ou méthodes sont visibles dans...	public	protected	"package"	private
...la même classe	Oui	Oui	Oui	Oui
...une classe du même package	Oui	Oui	Oui	Non
...une sous-classe d'un autre package	Oui	Oui	Non	Non
...une classe quelconque d'un autre package	Oui	Non	Non	Non

La portée : Classe et Interface

- public
  Elle est visible de toutes les autres classes
- ~~protected~~ (sauf pour les inner class)
- Sans mot clé mais dit “package”
  Elle est visible que dans le package
- ~~private~~ (sauf pour les inner class)

La portée : Constructeur

- public
  Il est visible de toutes les autres classes
- protected
  Il est visible par ses sous-classes (même d'un autre package) ou classes du même package
- Sans mot clé mais dit "package"
  Il est visible des **autres classes du même package**
- private
  Elle n'est visible qu'au sein de la classe.
  C'est indispensable avec le Design Pattern Singleton.

La portée : Attribut

- public
  Il est visible de toutes les autres classes
- protected
  Il est visible par ses sous-classes (même d'un autre package) ou classes du même package
- Sans mot clé mais dit "package"
  Il est visible des **autres classes du même package**
- private
  Il n'est visible qu'au sein de la classe

La portée : Méthode

- public
  Elle est visible de toutes les autres classes
- protected
  Elle est visible par ses sous-classes (même d'un autre package) ou classes du même package
- Sans mot clé mais dit "package"
  Elle est visible des **autres classes du même package**
- private
  Elle n'est visible qu'au sein de la classe

Remarque : Les droits d'accès des méthodes redéfinies ne doivent pas être diminués !
Une méthode public dans la super-classe ne peut pas être redéfinie private dans ses sous-classes.
L'inverse est cependant possible.

TD04.10-La Portée

Exercice :

1. Dans votre IDE, créez un nouveau projet Java nommé par exemple “AccessControl”
2. Définissez un nom de package
3. Créez une autre classe nommée Truc qui aura 4 attributs (aux noms significatifs) de portée différentes et 4 méthodes (aux noms significatifs) de portée différente
4. Créez une autre classe nommée TrucSpecial dans le même package qui hérite de Truc
   Codez une méthode foo() qui essaiera d’utiliser les attributs et méthodes de Truc
   Que constatez-vous ?
5. Créez un nouveau package et copiez-y la classe TrucSpecial.
   Que constatez-vous ?

Association de classes

Généralités

Etant donné qu'en POO, les objets interagissent entre eux, il existe donc des relations entre leurs classes.
On peut citer les relations suivantes :

- Héritage (que l'on a vu) -> relation de type "est un"
- Association -> relation entre classes de type "a un" ou "a plusieurs"
- Agrégation -> relation de type "se compose de"
- Composition -> agrégation entre objets d'un même cycle de vie
- Dépendance

Questions/Réponses