Cours 3

Introduction au langage Java

Rappel

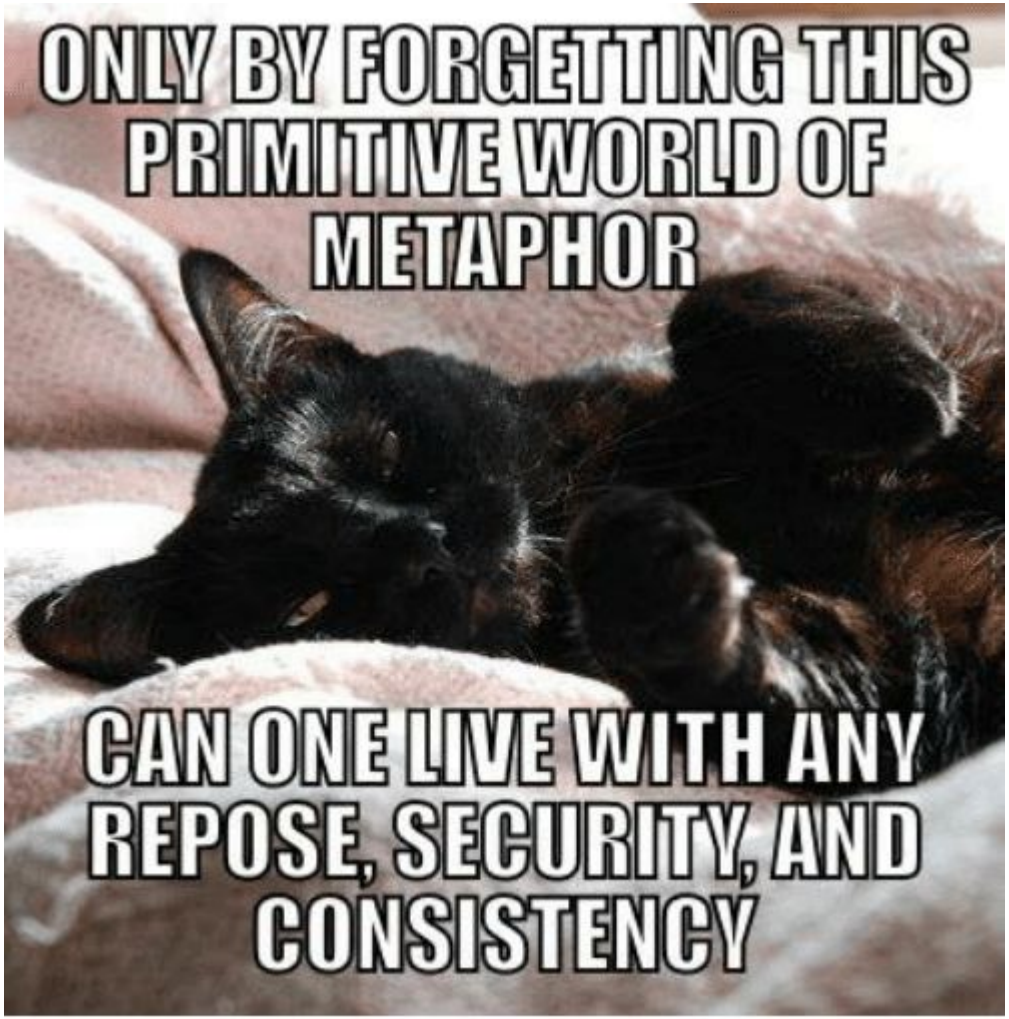
- Programmation orientée objet
  - Classe
  - Interface
  - Paquet
  - Modificateurs d’accès
  - Instance vs. statique
  - Méthode vs. fonction
  - Propriété (variable, accesseurs)
  - Héritage

Particularités du Java

- Syntaxe *similaire* au C#
- Entièrement orienté objet
- Héritage simple
- Interfaces multiples
- Typage statique et fort
- Toutes les classes héritent ultimement de **java.lang.Object**
- Chaque classe publique doit être dans un fichier du même nom dans un répertoire correspondant au paquet

Types primitifs

- byte
- short
- int
- long
- float
- double
- boolean
- char



- Toutes les chaînes de caractères sont des instances de la classe **java.lang.String**.
- Les tableaux sont également des objets mais leur classe n’est pas accessible directement.

Modificateurs d'accès

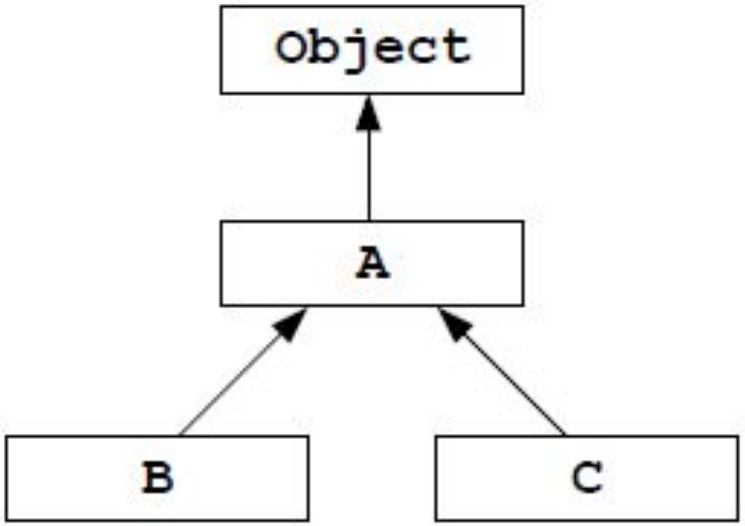
- Il y a 4 modificateurs d'accès en Java:
  - **public**
  - **protected**
  - *aucun modificateur*
  - **private**
- Ces modificateurs affectent l'accès global, par les sous-classes, par les autres classes du même paquet, et l'accès local.

	Classe	**Paquet**	**Sous-classes**	**Global**
public	Oui	Oui	Oui	Oui
protected	Oui	Oui	Oui	Non
aucun modificateur	Oui	Oui	Non	Non
private	Oui	Non	Non	Non

Héritage

- Chaque classe peut hériter d'une seule classe parent à l'aide du mot **extends**.

```
class A
{
    // ...
}

class B extends A
{
    // ...
}

class C extends A
{
    // ...
}
```



Virtualité

- Toutes les méthodes en Java sont automatiquement “virtuelles”. Elles peuvent toutes être remplacées dans une sous-classe sauf si déclarée avec **final**.
- L’annotation “**@Override**” est optionnelle et sert uniquement pour détecter les erreurs.
- On peut distinguer entre les membres de la classe actuelle et ceux de la classe parent avec les mots **this** et **super**.

```java
class A
{
    void afficheOrigine()
    {
        System.out.println("provient de A");
    }
}

class B extends A
{
    void afficheOrigine()
    {
        System.out.println("provient de B");
    }
}

class C extends B
{
    void afficheOrigine()
    {
        System.out.println("provient de C");
    }
}
```

```java
// plus loin dans C

afficheOrigine();
// Affiche: provient de C

this.afficheOrigine();
// Affiche: provient de C

super.afficheOrigine();
// Affiche: provient de B

super.super.afficheOrigine();
// ERREURE DE COMPILATION
```

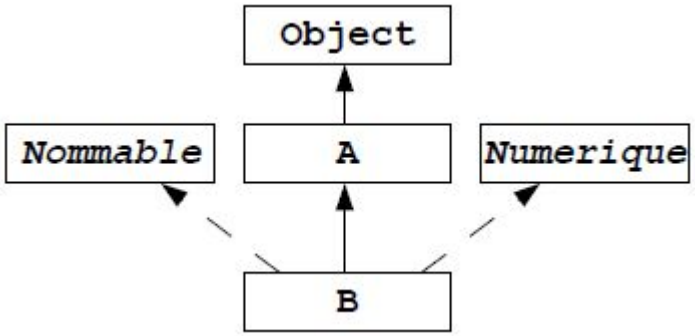

STOP!
I has a headache!
ICANHASCHEEZBURGER.COM

Interfaces

- Une classe peut implémenter plusieurs interfaces à l'aide du mot **implements**.

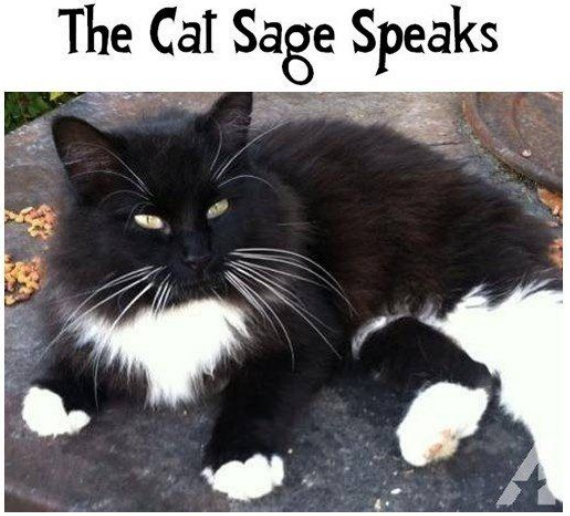
```java
interface Nommable
{
    void renomme(String nom);

    void afficheNom();
}

interface Numerique
{
    int intValue();
}

class A
{
    // ...
}
```

```java
class B extends A implements Nommable, Numerique
{
    int valeur = 0;

    String nom;

    public int intValue()
    {
        return valeur;
    }

    public void renomme(String nom)
    {
        this.nom = nom;
    }

    public void afficheNom()
    {
        System.out.println(nom);
    }
}
```



Classes imbriquées

- Une classe imbriquée (*nested class*) est une classe définie à l'intérieur d'une autre classe.
- Tout comme les variables et méthodes, une classe imbriquée peut être définie à deux niveaux:
  - Statique
  - Instance (*inner class*)



- Les classes imbriquées statiques peuvent être utilisées comme toutes autres classes.
- Contrairement au classes régulières, elles peuvent se faire assigner tous les modificateurs d'accès.
- La classe imbriquée fait partie de la classe externe, et a accès à tous ses éléments peu importe leurs modificateurs d'accès.

- La classe imbriquée statique peut accéder à tous les éléments statiques de la classe externe sans spécifier le nom de la classe.
- Si un élément porte le même nom, elle peut spécifier lequel en lui ajoutant le nom de classe.

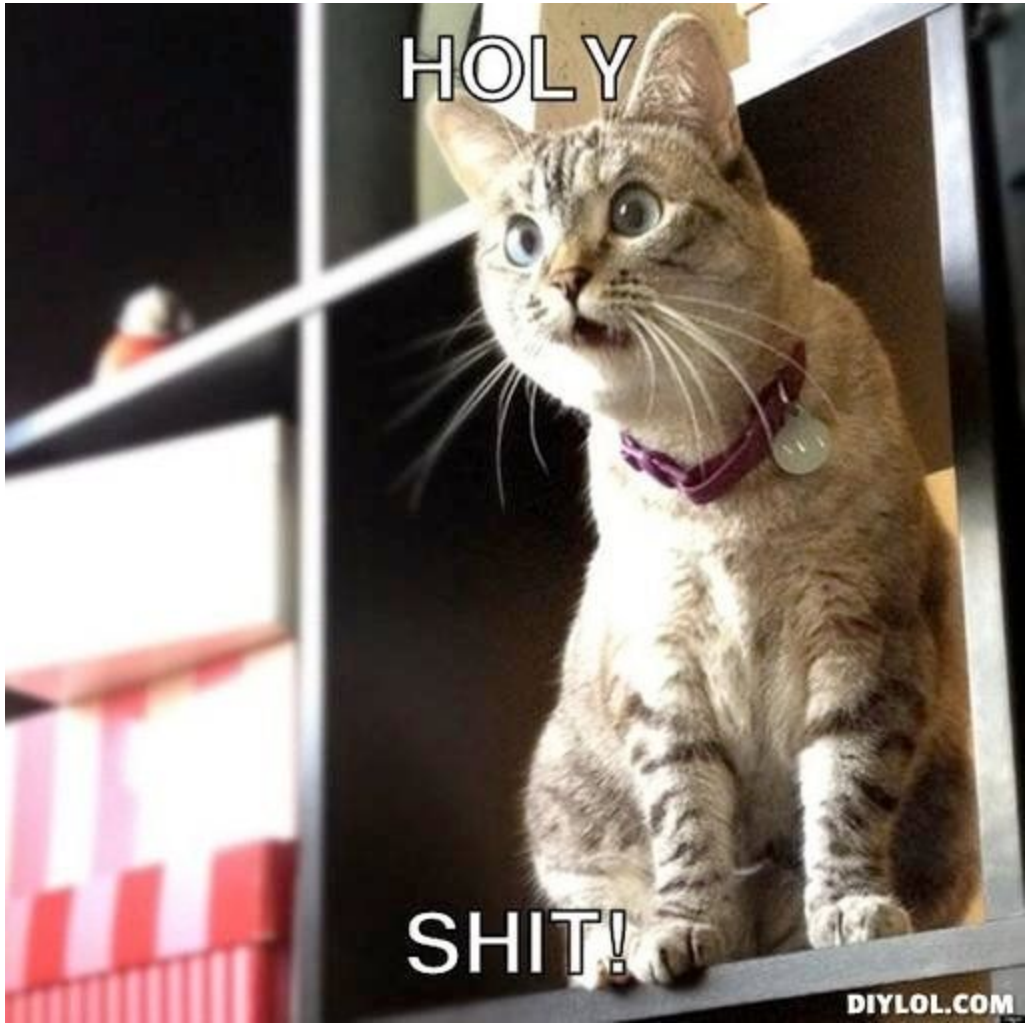
```java
class A
{
    static public int i = 0;
    static protected int j = 1;
    static int k = 2;
    static private int l = 3;

    static class B
    {
        static int i = 4;

        static void somme()
        {
            System.out.println(A.i + j + k + l + i);
            // Affiche: 10
        }
    }
}
```

```java
A.B.somme();
// Affiche: 10
```

- Une classe imbriquée statique peut également accéder aux éléments d'instance de la classe externe sur un objet donné.

```
class A
{
    static class B
    {
        static void afficheNom(A obj)
        {
            System.out.println(obj.nom);
        }
    }

    private String nom = "Mon nom";
}
```

```
A unObject = new A();
A.B.afficheNom(unObject);
// Affiche: Mon nom
```


HOLY
SHIT!
DIYLOL.COM

- Les classes imbriquées d'instance, appelées classes internes (*inner classes*), sont des classes dont tous les objet instanciés sont liés à une instance de la classe externe.
- Elles ne peuvent être instanciées que par un objet existant de la classe externe.

```
class A
{
    class B
    {
        void afficheNom()
        {
            System.out.println(nom);
        }
    }

    private B maClassInterne = new B();

    private String nom = "Mon nom";

    public void afficheNom()
    {
        maClassInterne.afficheNom();
        // Affiche: "Mon nom"
    }
}
```

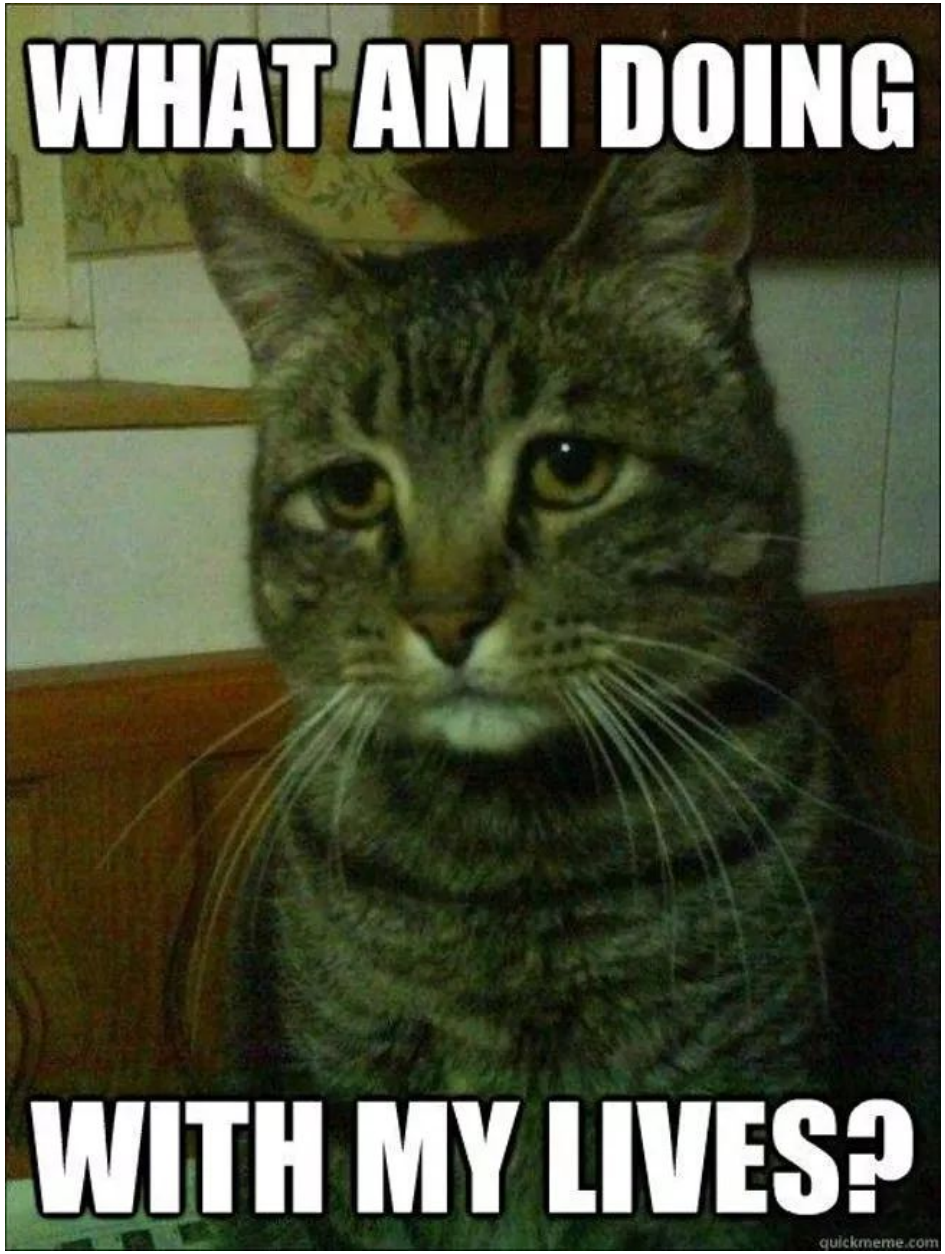
- Attention! Une classe interne n'est pas synonyme d'une sous-classe. On ne peut pas utiliser **super** pour accéder aux éléments de la classe externe, on doit utiliser **this** avec le nom de la classe.

```
class A
{
    class B
    {
        private String nom = "Classe interne";

        void afficheNom()
        {
            System.out.println(nom);
            // Affiche: Classe interne

            System.out.println(A.this.nom);
            // Affiche: Classe externe
        }
    }

    private String nom = "Classe externe";
}
```

- La classe externe et la classe interne peuvent avoir des classes parent différentes, et l'on peut précéder le mot clé **super** par le nom de la classe pour les distinguer.

```java
class A
{
    void afficheNom()
    {
        System.out.println("Class A");
    }
}

class B
{
    void afficheNom()
    {
        System.out.println("Class B");
    }
}
```

```java
class Externe extends A
{
    void afficheNom()
    {
        System.out.println("Class Externe");
    }

    class Interne extends B
    {
        void afficheNom()
        {
            Externe.this.afficheNom();
            // Affiche: Class Externe

            Externe.super.afficheNom();
            // Affiche: Class A

            super.afficheNom();
            // Affiche: Class B
        }
    }
}
```


WHAT AM I DOING
WITH MY LIVES?
quickmeme.com

- Il y existe deux sous-types additionnels de classes internes:
  - classes locales
  - classes anonymes

- Les classes locales sont définies à l'intérieur d'une méthode et ont accès aux variables locales ***finales*** de cette méthode.

```
class A
{
    void afficheNom()
    {
        System.out.println("Pas de nom");
    }
}

class MaClasse
{
    String prefix = "Ma Classe";

    A renomme(final String nom)
    {
        class B extends A
        {
            void afficheNom()
            {
                System.out.println(prefix + " " + nom);
            }
        }

        return new B();
    }
}
```

```
A[] liste = new A[3];
for (int i = 0; i < 3; ++i)
{
    liste[i] = renomme("Instance " + i);
}

liste[0].afficheNom();
// Affiche: Ma Classe Instance 0

liste[1].afficheNom();
// Affiche: Ma Classe Instance 1

liste[2].afficheNom();
// Affiche: Ma Classe Instance 2
```

- Les classes anonymes sont des classes qui sont définies sans être déclarées. La classe n'a pas de nom: sa définition est adjointe à une instanciation de sa classe parent (ou d'un interface).

```
A obj = new A()
{
    // ...
};
```

- ***ATTENTION***: On ne peut pas définir de constructeur dans une classe anonyme.

- Comme les classes locales, les classes anonymes ont accès aux éléments locaux du contexte où elles ont été instanciées.

```
interface A
{
    void afficheNom();
}
```

```
final String nom = "Mon nom";
A obj = new A()
{
    public void afficheNom()
    {
        System.out.println(nom);
    }
};
obj.afficheNom();
// Affiche: Mon nom
```