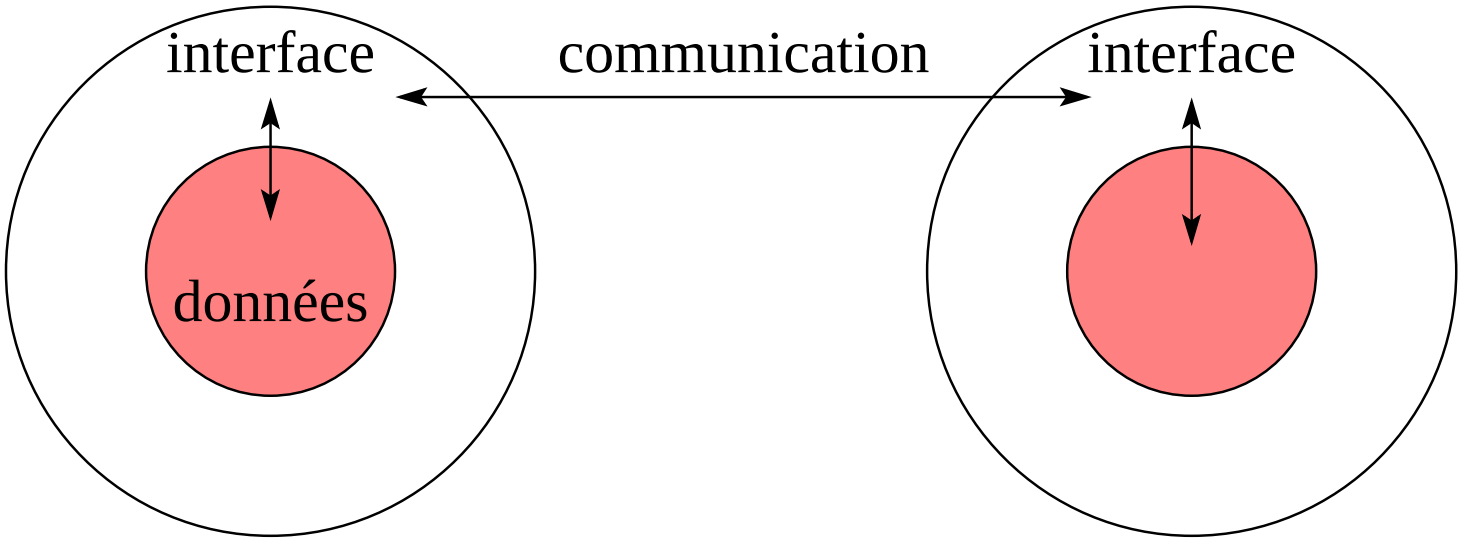
Java et OOP

Cours no. 2

Qu'est-ce que la OOP?



- Encapsulation, abstraction de données.
- Héritage = réutilisation du code.
- Polymorphisme.

Qu'est-ce qu'un objet ? Qu'est-ce qu'une classe ?

- *Attributs* (*champs, membres*) = données "privées".
- *Méthodes* = fonctions travaillant sur les données = interface pour l'objet pour communiquer avec son "environnement" (autres objets, utilisateur...)

```
monObjet.donnée=x;
monObjet.afficher()
```

- Classe = **type d'objets**.

```
class MaClasse monObjet;
int i;
```

Classes

Une classe `Cls` définit :

- le type de chaque attribut des objets dans `Cls`.
- le type de chaque paramètre pour chaque méthode qui appartient aux objets dans `Cls`, ainsi que le type de la valeur que ces méthodes rétournent.
- la définition (i.e. le *code*) de chaque méthode déclarée dans `Cls`.
- certains attributs communs pour tous les objets dans la classe `Cls`, ainsi que les méthodes qui les emploient.
- qu'est-ce qu'il faut faire quand on crée un objet de la classe `Cls`.
- qu'est-ce qu'il faut faire quand on veut éliberer la mémoire qu'un objet de type `class Cls` a utilisé mais qui n'est plus référencée.
- quelle est la portée des attributs et des méthodes appartenant aux objets de la classe `Cls` – s'ils sont accéssibles pour tout le monde, pour les "copains", pour les "héritiers" ou *seulement pour les objets du même type.*

Une classe

(Sans attributs ou méthodes "de classe", sans constructeurs)

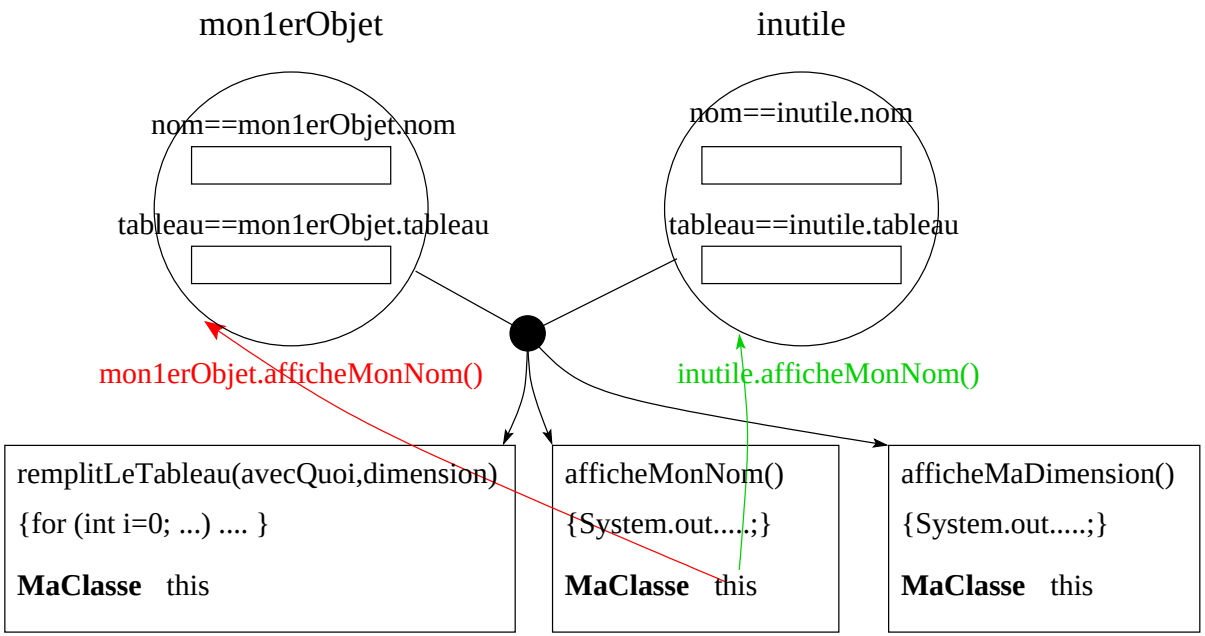
```
class MaClasse {
    float[] tableau;
    String nom;
    void afficheMaDimension() { System.out.println(tableau.length); }
    void afficheMonNom() { System.out.println("Je m'apelle " + nom); }
    public void remplitLeTableau(float avecQuoi, int dimension){
        tableau=new float[dimension];
        for (int i=0; i<tableau.length; i++)
            tableau[i]=avecQuoi;
    }
    public float renvoieAutreChose(){
        return tableau[0];
    }
}
```

- Possibilité d'imbriquer des classes dans d'autres : **encapsulation**.

A qui appartiennent les attributs et les méthodes?

```
class MaClasseAuTravail {
    public static void main (String[] args) {
        MaClasse mon1erObjet = new MaClasse;
        MaClasse inutile = new MaClasse;
        mon1erObjet.nom = new String("coucou");
        inutile.nom = new String("vraiment inutile");
        mon1erObjet.remplitLeTableau(2.0,10);
        nom1erObjet.afficheMonNom();
        inutile.afficheMonNom();
    }
}
```

A qui appartiennent ...?



Pour l'appel

`mon1erObjet.afficheMonNom()`,

nous avons

`this == mon1erObjet`

Dans ce cas,

```
nom == this.nom ==
    == mon1erObjet.nom
```

L'utilisation explicite de la référence `this`

Est parfois nécéssaire :

```
class AutreClasseInutile {
    int attr;
    public void modifie(int attr) {
        this.attr = attr;
    }
}
```

Surcharge des méthodes

- On peut avoir des méthodes ayant *le même nom*, mais avec des *listes des paramètres de types différents* et/ou du *type de retour différent*

```
class C {
    int[] x;
    int calculeQqChose();
    void calculeQqChose(int y);
}
```

- Utile pour la gestion des noms dans des grands programmes.
- **Intérdite** si la différence n'est que dans le type de retour !

Constructeurs

Traitement unifié de la création et de l'initialisation d'un objet.

```
class MaClasseAvecConstr {
    float[] tableau;
    String nom;
    void afficheMaDimension() { System.out.println(tableau.length); }
    void afficheMonNom() { System.out.println("Je m'apelle " + nom); }
    public MaClasseAvecConstructeur(float avecQuoi, int dimension){
        tableau=new float[dimension];
        for (int i=0; i<tableau.length; i++)
            tableau[i]=avecQuoi;
            nom = new String("Implicite");
    }
}
class MaClasseAuTravail {
    public static void main(String[] args) {
        MaClasseAvecConstr nouveauObjet = new MaClasseAvecConstr(2.0,4);
    }
}
```

Constructeurs(2)

- Le constructeur est appelé au moment de la demande d'allocation `new`.
- On peut *surcharger* les constructeurs.
- Si nous n'avons pas écrit nous-mêmes de constructeur, le langage génère un *par défaut*, sans paramètres et qui initialise tous les attributs par leurs valeurs par défaut (ex. `String` avec `null`).
- Le constructeur par défaut n'est plus généré si nous avons écrit nous-mêmes au moins un constructeur, *même avec des paramètres.*
- Autre façon d'appeler un constructeur : *dans un autre constructeur*:

```
class C{
    public C(int z) {....}
    public C() { this(2); } // j'appelle le constructeur C(int)
}
```

Les types des paramètres identifient le constructeur appelé.

Objets et références

- Les objets sont passés *toujours par référence.*
- Un tableau d'objets est en fait *un tableau de références* vers les objets – il faut allouer chaque entrée du tableau :

```
MaClasse[] tab = new MaClasse[10]; // j'ai alloué 10 places pour des
                                   // références aux objets MaClasse
for(int i=0;i<10;i++)
    MaClasse[i] = new MaClasse();  // seulement maintenant j'alloue
                                   // de l'espace pour mes objets
```

- Il n'est pas nécéssaire de désallouer un objet – *garbage collection* quand il n'y a plus de référence vers l'espace résérvé à l'objet.

Attributs et méthodes de classe (statiques)

- Attributs qui n'appartiennent pas à un objet – donnent des caractéristiques pour toute la classe.
  Exemple : nb. d'objets de classe `Cls` qu'on a créé.
- Méthodes qui utilisent ces attributs : e.g. gérer le nb. d'objets de classe `Cls`.

```
class Cls {
    static noObjets=0;
    static incrementeNoObjets() { noObjets++; }
    static afficheNoObjets() {System.out.println(noObjets); }
}
class UtiliseCls {
    public static void main(String[] args) {
        Cls.afficheNoObjets();       // bien-sur, ce sera 0
    }
}
```

Règles d'emploi des statiques

- Utilisation d'attribut ou appel de méthode statique : on préfixe avec le *nom de la classe* !
- Il n'y a pas de `this` dans une méthode statique !
- On ne peut pas appeler des méthodes d'instance dans des méthodes statiques, ni d'utiliser/modifier des attributs d'instance.
- *Blocs d'initialisation* statique (aussi pour les attributs d'instance).
- La méthode `main` doit être statique !

Constantes

- Constante d'instance : `final int x=5;`
- Constante de classe : `static final y;`
- Intialisation dans
  - un bloc d'initialisation,
  - un constructeur (pour les non-statiques).
- Paramètres finaux – non-modifiables à l'intérieur de la méthode.

Les méthodes peuvent être elles aussi finales, de même que les classes – voir héritage.

Paquetages

- Classes – organisées en **paquetages**.
- Utilisation de la classe Y du paquetage `x` : `x.Y`
- A travers le web :
  `fr.univ-paris12.babbage.dima.java.UneClasse`
  pour la classe `UneClasse` qui se trouve à
  `http://babbage.univ-paris12.fr/~dima/java`
- Nom court : seulement `UneClasse` dans le programme, si
  `import fr.univ-paris12.babbage.dima.java.*`

Règles de visibilité

- `public` – tout le monde a acces à cet attribut/cette méthode, y compris ceux qui ne sont pas dans le même paquetage.
- `private` – que les objets de la classe qui déclare cet attribut/cette méthode ont acces à celui-ci.
- *implicit* (classe de visibilité *paquetage*) – toutes les méthodes des autres classes *dans le même paquetage* ont acces.
- `protected` – en relation avec l'héritage.
- Les classes aussi peuvent être publiques ou privées.
- Les constructeurs aussi peuvent être publics, privés, protégés ou dans la classe de visibilité *paquetage*.
- Au maximum, une seule classe ou interface par paquetage peut être *publique* – declaration explicite. S'il y en a une, le nom du fichier doit être celui de la classe publique.

L'esprit de la programmation objet

- Les attributs ne devraient être visibles que pour les objets de la même classe.
- La communication entre objets de classes différentes se fait par l'intermédiaire des méthodes.

Attributs *privés*, méthodes *publiques* – sauf pour les méthodes à usage interne d'une classe.

- Chaque méthode publique devrait être "spécifiée" :

```
public String maMethode(int param){
// Requiert : contraintes sur les parametres/attributs
// Modifie : certains parametres ou attributs
// Effet : qu'est-ce qu'elle fait/renvoie
```