

Initiation à la programmation Java - IP2

Rappels du cours No 1

Giulio Manzonetto



Rappels

- Un nom de classe commence par une majuscule





Rappels

- Un **nom de classe** commence par une majuscule
- Le **code** s'écrit dans un fichier `.java` de même nom que la classe





Rappels

- Un **nom de classe** commence par une majuscule
- Le **code** s'écrit dans un fichier .java de même nom que la classe
- Les **attributs** (ou champs, ou propriétés) des objets sont déclarés dans le corps de la classe





Rappels

- Un **nom de classe** commence par une majuscule
- Le **code** s'écrit dans un fichier .java de même nom que la classe
- Les **attributs** (ou champs, ou propriétés) des objets sont déclarés dans le corps de la classe
- Les **variables nommant des objets** ont pour valeurs des **références** (ou pointeurs)



Rappels

- Un **nom de classe** commence par une majuscule
- Le **code** s'écrit dans un fichier .java de même nom que la classe
- Les **attributs** (ou champs, ou propriétés) des objets sont déclarés dans le corps de la classe
- Les **variables nommant des objets** ont pour valeurs des références (ou pointeurs)
- Le couple **new/constructeur** permet de construire les objets

Rappels

- Le **code** s'écrit dans un fichier .java de même nom que la classe
- Les **attributs** (ou champs, ou propriétés) des objets sont déclarés dans le corps de la classe
- Les **variables nommant des objets** ont pour valeurs des références (ou pointeurs)
- Le couple **new/constructeur** permet de construire les objets
- Un **constructeur** est une sorte de méthode :
 - qui n'est pas statique,
 - qui n'a pas besoin de type retour
 - qui porte le même nom que la classe
 - plusieurs constructeurs sont possibles
(si les arguments sont de signatures différentes)
- Constructeur non défini \iff on utilise un constructeur par défaut.

Rappels

- Les **attributs** (ou champs, ou propriétés) des objets sont déclarés dans le corps de la classe
- Les **variables nommant des objets** ont pour valeurs des références (ou pointeurs)
- Le couple **new/constructeur** permet de construire les objets
- Un **constructeur** est une sorte de méthode :
 - qui n'est pas statique,
 - qui n'a pas besoin de type retour
 - qui porte le même nom que la classe
 - plusieurs constructeurs sont possibles (si les arguments sont de signatures différentes)
- Constructeur non défini \iff on utilise un constructeur par défaut.
- On accède à un champ *x* d'un objet *a* en écrivant *a.x*

Rappels

- Les **variables nommant des objets** ont pour valeurs des références (ou pointeurs)
- Le couple **new/constructeur** permet de construire les objets
- Un **constructeur** est une sorte de méthode :
 - qui n'est pas statique,
 - qui n'a pas besoin de type retour
 - qui porte le même nom que la classe
 - plusieurs constructeurs sont possibles
(si les arguments sont de signatures différentes)
- Constructeur non défini \iff on utilise un constructeur par défaut.
- On accède à un champ *x* d'un objet *a* en écrivant *a.x*
- L'appel d'une méthode **statique *f()*** écrite dans une classe *C* se fait en utilisant la syntaxe *C.f()*

Rappels



- Le couple **new/constructeur** permet de construire les objets
- Un **constructeur** est une sorte de méthode :
 - qui n'est pas statique,
 - qui n'a pas besoin de type retour
 - qui porte le même nom que la classe
 - plusieurs constructeurs sont possibles
(si les arguments sont de signatures différentes)
- Constructeur non défini \iff on utilise un constructeur par défaut.
- On accède à un champ *x* d'un objet *a* en écrivant *a.x*
- L'appel d'une méthode **statique f()** écrite dans une classe *C* se fait en utilisant la syntaxe *C.f()*
- **Rappel !** Avez vous installé Eclipse ou Netbeans chez vous ?

Exercice : retrouver les rappels précédent sur cet exemple

fichier : Triangle.java

```
public class Triangle{  
    Point [] tab;  
    Triangle(Point a,Point b,Point c){  
        tab = new Point[3];  
        tab[0] = a;  
        tab[1] = b; tab[2] = c;  
    }  
}
```

fichier : Test.java

```
public class Test{  
    public static void main(String[]  
        args){  
        Point p = new Point(0,0);  
        Cercle c = new Cercle(p,5);  
        int d = Cercle.diametre(c);  
    }  
}
```

fichier : Cercle.java

```
public class Cercle{  
    Point centre;  
    int rayon;  
    Cercle (int a, int b, int c){  
        p = new Point(a,b);  
        rayon = c;  
    }  
    Cercle (Point x, int d){  
        centre = x;// référence partagée  
        rayon = d;  
    }  
    static int diametre(Cercle c){  
        return 2 * c.rayon;  
    }  
}
```

fichier : Point.java

Initiation à la programmation Java

IP2 - Cours No 2

Giulio Manzonetto

Static or not Static ?

On a évoqué deux notations :

- `MaClasse.maMethode()`

accès à la méthode `maMethode()`

écrite statiquement dans la classe `MaClasse`

- `monObjet.sonContenu`

accès au champ `sonContenu`

d'un objet particulier `monObjet`



Static or not Static ?



On a évoqué deux notations :

- `MaClasse.maMethode()`
accès à la méthode `maMethode()`
écrite statiquement dans la classe `MaClasse`
- `monObjet.sonContenu`
accès au champ `sonContenu`
d'un objet particulier `monObjet`

Et bien il existe tous les cas de figures :

- `MaClasse.sonPropreContenu`
- `monObjet.maPropreMethode()`

La différence de signification est liée à l'absence ou à la présence du mot clé **static**

Static or Not Static ?

données de classe ou données d'instance

fichier : Exemple.java

```
public class Exemple{  
    static int nbInstance = 0;  
    String info;  
    Exemple(String arg){ info = arg; nbInstance++; }  
}
```

fichier : Test.java

```
public class Test{  
    public static void main(String[] args){  
        Exemple e1 = new Exemple ("premier");  
        System.out.println(Exemple.nbInstance); // affiche 1  
        Exemple e2 = new Exemple ("second");  
        System.out.println(Exemple.nbInstance); // affiche 2  
    }  
}
```

fichier : Exemple.java

```
public class Exemple{  
    static int nbInstance = 0;  
    String info;  
    Exemple(String arg){ info = arg; nbInstance++; }  
}
```

fichier : Test.java

```
public class Test{  
    public static void main(String[] args){  
        Exemple e1 = new Exemple ("premier");  
        System.out.println(Exemple.nbInstance); // affiche 1  
        Exemple e2 = new Exemple ("second");  
        System.out.println(Exemple.nbInstance); // affiche 2  
    }  
}
```

- La variable statique n'est pas directement liée à e1 ou à e2
- Elle est liée à l'ensemble de ces objets (la classe) c.-à-d. partagée

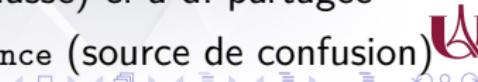
fichier : Exemple.java

```
public class Exemple{  
    static int nbInstance = 0;  
    String info;  
    Exemple(String arg){ info = arg; nbInstance++; }  
}
```

fichier : Test.java

```
public class Test{  
    public static void main(String[] args){  
        Exemple e1 = new Exemple ("premier");  
        System.out.println(Exemple.nbInstance); // affiche 1  
        Exemple e2 = new Exemple ("second");  
        System.out.println(Exemple.nbInstance); // affiche 2  
    }  
}
```

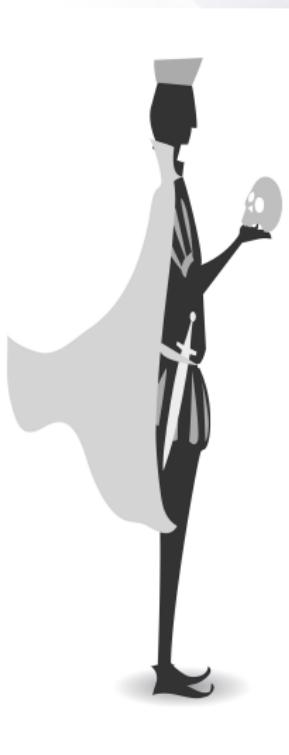
- La variable statique n'est pas directement liée à e1 ou à e2
- Elle est liée à l'ensemble de ces objets (la classe) c.-à-d. partagée
- Rq : on peut aussi y accéder par e1.nbInstance (source de confusion)



Static or Not Static ?

On a donc vu :

- MaClasse.maMethode()
- monObjet.sonContenu
- MaClasse.sonPropreContenu



Static or Not Static ?



On a donc vu :

- `MaClasse.maMethode()`
accès à la méthode `maMethode()`
écrite statiquement dans la classe `MaClasse`
- `monObjet.sonContenu`
accès au champ `sonContenu`
d'un objet particulier `monObjet`
- `MaClasse.sonPropreContenu`
accès au champ unique `sonPropreContenu`, partagé
par tous les objets de `MaClasse`
Défini **static**

Static or Not Static ?



On a donc vu :

- `MaClasse.maMethode()`
accès à la méthode `maMethode()`
écrite statiquement dans la classe `MaClasse`
- `monObjet.sonContenu`
accès au champ `sonContenu`
d'un objet particulier `monObjet`
- `MaClasse.sonPropreContenu`
accès au champ unique `sonPropreContenu`, partagé
par tous les objets de `MaClasse`
Défini **static**

Il reste à voir :

- `monObjet.maPropreMethode()`

Static or Not Static ?

Une formulation indirecte/directe (ce semestre)

- Une phrase du langage usuel se décompose en :



exemple : "JE MANGE UN STEAK DE BOEUF"

- Une forme **statique** n'aura pas de sujet.
Elle sera juste de la forme : "verbe + paramètres"
ici on dirait : "MANGER MOI STEAK BOEUF"
- On passe d'un style à l'autre en distinguant un **responsable de l'action**.

Static or Not Static ?

Une formulation indirecte/directe (ce semestre)

- Une phrase du langage usuel se décompose en :



exemple : "JE MANGE UN STEAK DE BOEUF"

- Une forme **statique** n'aura pas de sujet.
Elle sera juste de la forme : "verbe + paramètres"
ici on dirait : "MANGER MOI STEAK BOEUF"

manger(moi,steak,boeuf)

- On passe d'un style à l'autre en distinguant un **responsable de l'action**.



Static or Not Static ?

Une formulation indirecte/directe (ce semestre)

- Une phrase du langage usuel se décompose en :



exemple : "JE MANGE UN STEAK DE BOEUF"

```
je.mange(steak,boeuf)
```

- Une forme **statique** n'aura pas de sujet.
Elle sera juste de la forme : "verbe + paramètres"
ici on dirait : "MANGER MOI STEAK BOEUF"

```
manger(moi,steak,boeuf)
```

- On passe d'un style à l'autre en distinguant un **responsable de l'action**.



Static or Not Static ?

Une formulation indirecte/directe (ce semestre)

- Une phrase du langage usuel se décompose en :



- Une méthode statique n'a pas de sujet. Juste "verbe + paramètres"

fichier : Point.java

```
public class Point{  
    int x,y; // chaque objet possède ses informations propres  
    public static double donneDistance(Point a, Point b){  
        int dx = b.x - a.x; // qu'on lit explicitement  
        int dy = b.y - a.y;  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
}
```

- En programmation objet, l'un des "compléments" devient acteur.
On cherche à le "responsabiliser".



- Une méthode non statique est exécutée par un objet "responsable"

fichier : Point.java

```
public class Point{  
    int x,y; // chaque objet possède ses informations propres  
    public double donneDistance(Point b){ // ici méthode non statique  
        int dx = x - b.x; // x et y sont les champs de l'objet "responsable"  
        int dy = y - b.y; // b.x et b.y sont ceux du paramètre b  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
}
```

- L'appel d'une méthode non statique se fait sur l'objet désigné responsable

fichier : Test.java

```
public class Test{  
    public static void main(String[] args){  
        Point toi = new Point (10,20), lui = new Point (50,100);  
        System.out.println( toi.donneDistance(lui) );  
    }  
}
```

On peut même faire coexister les deux versions (car signatures distinctes) :

fichier : Point.java

```
public class Point{  
    int x,y;  
    public double distance(Point b){ // non statique  
        int dx = x - b.x;  
        int dy = y - b.y;  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
    public static double distance(Point a, Point b){ // statique  
        return a.distance(b);  
    }  
}
```

fichier : Test.java

```
public class Test{  
    public static void main(String[] args){  
        Point toi = new Point (10,20), lui = new Point (50,100);  
        System.out.println(toi.distance(lui)); // appel non statique  
        System.out.println(Point.distance(toi,lui)); // appel statique  
    }  
}
```

Static or not Static ?

Résumé des 4 écritures :

- MaClasse.maMethode()
- monObjet.sonContenu
- MaClasse.sonPropreContenu
- monObjet.maPropreMethode()



Static or not Static ?

Résumé des 4 écritures :

- `MaClasse.maMethode()`
accès à la méthode `maMethode()`,
écrite statiquement dans la classe `MaClasse`
- `monObjet.sonContenu`
accès au champ `sonContenu`,
d'un objet particulier `monObjet`
- `MaClasse.sonPropreContenu`
accès au champ unique `sonPropreContenu`, partagé
par tous les objets de `MaClasse`
- `monObjet.maPropreMethode()`
`monObjet` est désigné responsable de l'exécution de
`maPropreMethode` qui est écrite non statique



Static or not Static ?



Résumé des 4 écritures :

- MaClasse.maMethode()
- monObjet.sonContenu
- MaClasse.sonPropreContenu
- monObjet.maPropreMethode()

Illustration : comment comprendre

```
System.out.println();
```

Static or not Static ?

Résumé des 4 écritures :

- MaClasse.maMethode()
- monObjet.sonContenu
- MaClasse.sonPropreContenu
- monObjet.maPropreMethode()

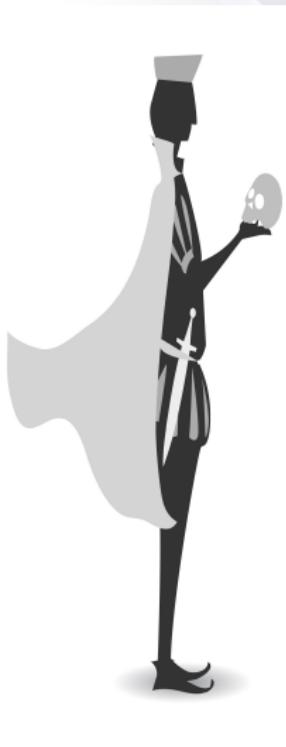
Illustration : comment comprendre

`System.out.println();`

- `System` apparaît comme étant une classe
- `out` apparaît comme étant un champ de classe (c.-à-d. un attribut `static`)
- `println()` apparaît comme étant une méthode non statique écrite dans la classe qui définit les objets de même type que `out`



Static or not Static ?



Résumé des 4 écritures :

- MaClasse.maMethode()
- monObjet.sonContenu
- MaClasse.sonPropreContenu
- monObjet.maPropreMethode()

Comment comprendre :

```
public class Test{  
    public static void main(String[] args){ ..etc..}  
}
```

Static or not Static ?



Résumé des 4 écritures :

- MaClasse.maMethode()
- monObjet.sonContenu
- MaClasse.sonPropreContenu
- monObjet.maPropreMethode()

Comment comprendre :

```
public class Test{  
    public static void main(String[] args){ ..etc.. }  
}
```

- **public** car `main` sera appelée de l'extérieur par Java (on reviendra sur ce mot clé)
- si **static** était enlevé, cela signifierait qu'il faudrait d'abord construire un objet `Test` responsable d'une exécution de `main`, ce qui compliquerait inutilement.

Static or not Static ?

interprétation selon le contexte

Remarque :

en Java on autorise, dans le code, des appels de la forme `uneMethode()` ; qui ne posent pas de problèmes dès lors que **le contexte est clair.**

- Si l'appel est situé dans un bloc de code `static`, il n'y a aucun objet responsable, cette méthode est donc recherchée parmi les méthodes `static`
Ici, aucune classe n'est précisée : cette méthode `static` doit être définie dans la classe actuelle.
- Si l'appel est situé dans un bloc de code non `static`, alors il y a un objet responsable, et la méthode est recherchée parmi les méthodes non statiques **ou** statiques. (\Rightarrow Il ne faut pas écrire d'ambiguïté)

Le mot clé this (1)

fichier : Point.java

```
public class Point{  
    int x,y;  
    public static double distance(Point a, Point b){ // version statique  
        int dx = a.x - b.x;  
        int dy = a.y - b.y;  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
    public double distance(Point b){ // version non statique  
        return distance(???,b); // appel à la méthode statique  
    }  
}
```

Le mot clé this (1)

fichier : Point.java

```
public class Point{  
    int x,y;  
    public static double distance(Point a, Point b){ // version statique  
        int dx = a.x - b.x;  
        int dy = a.y - b.y;  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
    public double distance(Point b){ // version non statique  
        return distance(this, b); // appel à la méthode statique  
    }  
}
```

Le mot clé this (2)

Résoudre un pb : une variable locale *x* peut masquer un champ *x*

fichier : Point.java - version initiale

```
public class Point{  
    int x,y;  
    public double distance(Point b){ // non statique  
        int dx = x - b.x;  
        int dy = y - b.y;  
        return Math.sqrt(dx*dx + dy*dy);  
    }  
}
```

Le mot clé this (2)

Résoudre un pb : une variable locale *x* peut masquer un champ *x*

fichier : Point.java - Non correct

```
public class Point{  
    int x,y;  
    public double distance(Point b){ // non statique  
        int x = x - b.x; // maladroit : x et y déclarées variables locales  
        int y = y - b.y; // cachent les attributs  
        return Math.sqrt(x*x + y*y);  
    }  
}
```

Le mot clé this (2)

Résoudre un pb : une variable locale *x* peut masquer un champ *x*

fichier : Point.java - Problème résolu

```
public class Point{  
    int x,y;  
    public double distance(Point b){ // non statique  
        int x = this.x - b.x; // les deux x coexistent sans ambiguïté  
        int y = this.y - b.y; // idem pour y  
        return Math.sqrt(x*x + y*y);  
    }  
}
```

Le mot clé this (3)

- **this** sert aussi pour réutiliser un constructeur précédemment écrit :

fichier : Cercle.java - Version possible (ici on écrit tout deux fois)

```
public class Cercle{  
    Point centre;  
    int rayon;  
    Cercle (Point x, int r){  
        centre = x;  
        rayon = r;  
    }  
    Cercle (int a, int b, int c){  
        centre = new Point(a,b);  
        rayon = c;  
    }  
}
```

Le mot clé this (3)

- **this** sert aussi pour réutiliser un constructeur précédemment écrit :
fichier : Cercle.java - Ici on rappelle un autre constructeur

```
public class Cercle{  
    Point centre;  
    int rayon;  
    Cercle (Point x, int r){  
        centre = x;  
        rayon = r;  
    }  
    Cercle (int a, int b, int c){  
        this (new Point(a,b), c);  
    }  
}
```

- La syntaxe **this(arguments)** correspond à la signature de l'autre constructeur
- Possible seulement en toute première instruction

- Adoptez comme règle pendant qqs semaines d'écrire **this** à chaque fois qu'il est implicite, même s'il n'est pas indispensable.
- Idem pour l'appel d'une méthode statique ou la lecture d'un attribut statique : forcez vous à écrire le nom de la classe devant.
- Dès à présent écrivez la majorité de vos méthodes en adoptant une syntaxe non statique.

Terminologie

Une méthode non statique est aussi appelée "dynamique".

Ce terme prendra tout son sens avec l'héritage en L2.

Chapitre suivant : la vie privée des objets



La vie privée des objets

- **public** : caractérise les champs et méthodes qui sont accessibles quel que soit l'endroit où le code s'exécute (cette classe ou une autre)
- **private** : caractérise les champs et méthodes qui sont accessibles uniquement par du code exécuté au sein de leur classe.
- **final** : caractérise un champ dont la valeur ne pourra pas être changée après son initialisation
- pour les accès indirect, la politique de modification et d'accès aux champs est précisée par des méthodes dites **getter** et **setter**
 - ce ne sont pas des mots clés
 - notions plus ou moins formelles
 - par convention on présente dans une classe : d'abord les champs, suivis des constructeurs, suivis des **getter** et **setter**, suivis de méthodes plus complexes

Rq : par défaut les champs et méthodes peuvent être considérés **public**

La vie privée des objets

Illustration

- On s'intéresse à une population d'individus
- Ils ont tous la connaissance partagée de l'année courante
- Lorsqu'ils naissent leurs noms et leurs années de naissances sont fixés
- Dans notre modèle, le temps peut passer, une année après l'autre
- On peut connaître l'age d'une personne majeure,
mais on obtiendra -1 comme réponse dans le cas d'un mineur

La vie privée des objets

Illustration

- On s'intéresse à une population d'individus
- Ils ont tous la connaissance partagée de l'année courante
- Lorsqu'ils naissent leurs noms et leurs années de naissances sont fixés
- Dans notre modèle, le temps peut passer, une année après l'autre
- On peut connaître l'age d'une personne majeure,
mais on obtiendra -1 comme réponse dans le cas d'un mineur

fichier : Individu.java

```
public class Individu {  
    ...  
}
```

La vie privée des objets

Illustration

- Ils ont tous la connaissance partagée de l'année courante
- Lorsqu'ils naissent leurs noms et leurs années de naissances sont fixés
- Dans notre modèle, le temps peut passer, une année après l'autre
- On peut connaître l'age d'une personne majeure,
mais on obtiendra -1 comme réponse dans le cas d'un mineur

fichier : Individu.java

```
public class Individu {  
    static int annéeCourante = 2030; // connaissance partagée  
    // initialisée à 2030  
    ...  
}
```

La vie privée des objets

Illustration

- Lorsqu'ils naissent leurs noms et leurs années de naissances sont fixés
- Dans notre modèle, le temps peut passer, une année après l'autre
- On peut connaître l'age d'une personne majeure,
mais on obtiendra -1 comme réponse dans le cas d'un mineur

fichier : Individu.java

```
public class Individu {  
    static int annéeCourante = 2030; // connaissance partagée  
    final String nom; // attribut de chaque individu, définitif !  
    final int annéeNaissance; // attribut de chaque individu, définitif !  
    Individu(String nom) {  
        this.nom = nom; // this est indispensable (ambiguïté)  
        this.annéeNaissance = annéeCourante; // Rq : this est facultatif ici  
    }  
}
```

- Dans notre modèle, le temps peut passer, une année après l'autre
- On peut connaître l'age d'une personne majeure,
mais on obtiendra -1 comme réponse dans le cas d'un mineur

fichier : Individu.java

```
public class Individu {  
    static int annéeCourante = 2030; // connaissance partagée  
    final String nom; // attribut de chaque individu, définitif !  
    final int annéeNaissance; // attribut de chaque individu, définitif !  
    Individu(String nom) {  
        this.nom = nom; // this est indispensable (ambiguïté)  
        this.annéeNaissance = annéeCourante; // Rq : this est facultatif ici  
    }  
}
```

fichier : Test.java - Problème

```
public class Test{  
    public static void main(String[] args ){  
        Individu.annéeCourante += 100;  
    }  
}
```

- Dans notre modèle, le temps peut passer, une année après l'autre
- On peut connaître l'age d'une personne majeure,
mais on obtiendra -1 comme réponse dans le cas d'un mineur

fichier : Individu.java

```
public class Individu {  
    private static int annéeCourante = 2030; // connaissance partagée  
        privée  
    ...  
    public static void annéeSuivante(){ // Une forme de setter  
        Individu.annéeCourante++; // rq: Individu est facultatif ici  
    }  
}
```

fichier : Test.java

```
public class Test{  
    public static void main(String[] args ){  
        Individu.annéeSuivante(); // une méthode statique, évidemment  
    }  
}
```

- On peut connaître l'âge d'une personne majeure,
mais on obtiendra -1 comme réponse dans le cas d'un mineur

fichier : Individu.java

```
public class Individu {  
    final String nom;  
    final int annéeNaissance;  
    ...  
}
```

fichier : Test.java - Problème

```
public class Test{  
    public static void main(String[] args ){  
        Individu x = new Individu("toto");  
        System.out.println(x.annéeNaissance); // on trouvera son age...  
    }  
}
```

fichier : Individu.java

```
public class Individu {  
    private static int annéeCourante = 2030;  
    private final String nom;  
    private final int annéeNaissance; // attribut privé  
    public int getAge(){ // une forme de getter  
        int val = Individu.annéeCourante - this.annéeNaissance;  
        if (val < 18) return -1;  
        else return val;  
    }  
}
```

fichier : Test.java

```
public class Test{  
    public static void main(String[] args ){  
        Individu x = new Individu("toto");  
        System.out.println(x.getAge()); // information contrôlée  
    }  
}
```

La notion de **contrat**.

- Il est important que les fonctionnalités (publiques) d'un composant correspondent à ce qu'attend son environnement.
- La manière dont il les réalise (en privé) est secondaire
- Exemple : dans la relation entre un composant Véhicule et un environnement Conducteur, ce qui est important c'est que le véhicule réponde aux commandes
 - démarre
 - stop
 - accélère
 - ralenti
 - rouleXsecondes(10)
 - tourneDeXDegrés(90)

On déclare le contrat attendu dans une **interface**

Complément Vie Privée/Publique - Notion de contrat

fichier : Vehicule.java

```
public interface Vehicule{  
    public boolean démarre(int code); // retourne vrai si succès  
    public void stop();  
    public int accelere(); // retourne la vitesse courante  
    public int ralenti();  
    public boolean rouleXseconde(int t); // retourne si bon état  
    public void tourneDexDegres(int x);  
}
```

fichier : Automobile.java

```
public class Automobile implements Vehicule {  
    private String modèle;  
    private int codeClé;  
    private boolean estEnMarche;  
    public boolean démarre(int code){  
        if (code == this.codeClé) this.estEnMarche = true;  
        return estEnMarche;  
    }  
    ... et toutes les autres méthodes  
}
```

Complément Vie Privée/Publique - Notion de contrat

- Des contrats multiples sont possibles

fichier : GéoLocalisable.java

```
public interface GéoLocalisable{  
    public double donneLatitude();  
    public double donneLongitude();  
}
```

fichier : Automobile.java

```
public class Automobile implements Vehicule, GéoLocalisable {  
    // des champs doivent permettre de calculer latitude et longitude  
    public double donneLatitude(){  
        // ce calcul doit être réalisé !  
    }  
    ... et toutes les autres méthodes  
}
```

Complément Vie Privée/Publique - Notion de contrat

- Les interfaces sont utilisables comme des types

fichier : Automobile.java

```
public class Automobile implements Vehicule {  
    ...  
}
```

fichier : Test.java

```
public class Test{  
    public static void main(String[] args){  
        Vehicule v = new Automobile("Aston Martin");  
        v.demarre(1234);  
    }  
}
```

fichier : Automobile.java

```
public class Automobile implements Vehicule {  
    ...  
}
```

fichier : Moto.java

```
public class Moto implements Vehicule {  
    ...  
}
```

fichier : Test.java

```
public class Test{  
    public static void main(String[] args){  
        Vehicule [] garage = new Vehicule[2];  
        garage[0] = new Automobile("Aston Martin");  
        garage[1] = new Moto("Ducati");  
        garage[0].demarre(1234);  
        garage[1].demarre(4321);  
    }  
}
```

LES BONNES MANIERES



- L'objet comme sujet, acteur
- Respecter sa vie privée
- Définir le contrat d'interface
- Vous pourrez ainsi livrer un travail, une librairie, intelligible

Utilisation de librairies Java - Exemple de String

- Les difficultés sont surmontées par des complications internes (votre travail), invisibles à l'utilisateur
- Toutes les classes de la librairie Java (API) procèdent ainsi.
On trouve dans l'API de la classe `String` :
 - `String(char[] value)` : un constructeur parmi d'autres
 - `char charAt(int index)` : retourne le caractère à cette position
 - `boolean endsWith(String suffix)` teste si la chaîne termine par le suffixe spécifié
 - `static String valueOf(int i)` retourne la chaîne qui représente l'entier donné en argument.
- On est en mesure de comprendre comment les utiliser
(et peu importe comment elles sont implémentées par un ingénieur)

On trouve dans l'API de la classe `String` :

- `String(char[] value)` : un constructeur parmi d'autres
- `char charAt(int index)` : retourne le caractère à cette position
- `boolean endsWith(String suffix)` teste si la chaîne termine par le suffixe spécifié
- `static String valueOf(int i)` retourne la chaîne qui représente l'entier donné en argument.

fichier : `TestString.java`

```
public class TestString{  
    public static void main(String[] args){  
        char [] t= {'a','b','c'};  
        String s = new String(t); // une construction possible de "abc"  
        System.out.println("le second caractère est :" + ???);  
    }  
}
```

Utilisation de librairies Java - Exemple de String

On trouve dans l'API de la classe `String` :

- `String(char[] value)` : un constructeur parmi d'autres
- `char charAt(int index)` : retourne le caractère à cette position
- `boolean endsWith(String suffix)` test si la chaîne termine par le suffixe spécifié
- `static String valueOf(int i)` retourne la chaîne qui représente l'entier donné en argument.

fichier : `TestString.java`

```
public class TestString{  
    public static void main(String[] args){  
        char [] t= {'a','b','c'};  
        String s = new String(t); // une construction possible de "abc"  
        System.out.println("le second caractère est :" + s.charAt(1));  
    }  
}
```

Utilisation de librairies Java - Exemple de String

On trouve dans l'API de la classe String :

- `String(char[] value)` : un constructeur parmi d'autres
- `char charAt(int index)` : retourne le caractère à cette position
- `boolean endsWith(String suffix)` test si la chaîne termine par le suffixe spécifié
- `static String valueOf(int i)` retourne la chaîne qui représente l'entier donné en argument.

fichier : TestString.java

```
public class TestString{  
    public static void main(String[] args){  
        char [] t= {'a','b','c'};  
        String s = new String(t); // une construction possible de "abc"  
        System.out.println("le second caractère est :" + s.charAt(1));  
        System.out.println(s.endsWith("bc")); // affiche true  
    }  
}
```

Utilisation de librairies Java - Exemple de String

On trouve dans l'API de la classe String :

- `String(char[] value)` : un constructeur parmi d'autres
- `char charAt(int index)` : retourne le caractère à cette position
- `boolean endsWith(String suffix)` test si la chaîne termine par le suffixe spécifié
- `static String valueOf(int i)` retourne la chaîne qui représente l'entier donné en argument.

fichier : TestString.java

```
public class TestString{  
    public static void main(String[] args){  
        char [] t= {'a','b','c'};  
        String s = new String(t); // une construction possible de "abc"  
        System.out.println("le second caractère est :" + s.charAt(1));  
        System.out.println(s.endsWith("bc")); // affiche true  
        String cent = ??? valueOf(100) ???;  
    }  
}
```

Utilisation de librairies Java - Exemple de String

On trouve dans l'API de la classe String :

- `String(char[] value)` : un constructeur parmi d'autres
- `char charAt(int index)` : retourne le caractère à cette position
- `boolean endsWith(String suffix)` test si la chaîne termine par le suffixe spécifié
- `static String valueOf(int i)` retourne la chaîne qui représente l'entier donné en argument.

fichier : TestString.java

```
public class TestString{
    public static void main(String[] args){
        char [] t= {'a','b','c'};
        String s = new String(t); // une construction possible de "abc"
        System.out.println("le second caractère est :" + s.charAt(1));
        System.out.println(s.endsWith("bc")); // affiche true
        String cent = String.valueOf(100); // appel d'une méthode statique
    }
}
```

Utilisation de librairies Java - Exemple de String

On trouve dans l'API de la classe `String` :

- `String(char[] value)` : un constructeur parmi d'autres
- `char charAt(int index)` : retourne le caractère à cette position
- `boolean endsWith(String suffix)` test si la chaîne termine par le suffixe spécifié
- `static String valueOf(int i)` retourne la chaîne qui représente l'entier donné en argument.

fichier : `TestString.java`

```
public class TestString{  
    public static void main(String[] args){  
        char [] t= {'a','b','c'};  
        String s = new String(t); // une construction possible de "abc"  
        System.out.println("le second caractère est :" + s.charAt(1));  
        System.out.println(s.endsWith("bc")); // affiche true  
        String cent = String.valueOf(100); // appel d'une méthode statique  
        System.out.println(cent+0); // affiche ???  
    }  
}
```

Utilisation de librairies Java - Exemple de String

On trouve dans l'API de la classe `String` :

- `String(char[] value)` : un constructeur parmi d'autres
- `char charAt(int index)` : retourne le caractère à cette position
- `boolean endsWith(String suffix)` test si la chaîne termine par le suffixe spécifié
- `static String valueOf(int i)` retourne la chaîne qui représente l'entier donné en argument.

fichier : `TestString.java`

```
public class TestString{  
    public static void main(String[] args){  
        char [] t= {'a','b','c'};  
        String s = new String(t); // une construction possible de "abc"  
        System.out.println("le second caractère est :" + s.charAt(1));  
        System.out.println(s.endsWith("bc")); // affiche true  
        String cent = String.valueOf(100); // appel d'une méthode statique  
        System.out.println(cent+0); // affiche 1000  
    }  
}
```

Utilisation de librairies Java - Exemple de Scanner

- autre exemple de l'API, Scanner : "*un scanner est un objet qui permet d'analyser un flux d'entrée (clavier, fichier, ...) en donnant la possibilité de le décomposer en mots*".

Avec une sélection de méthodes publiques :

- `Scanner(InputStream source)` : constructeur qui peut prendre en argument par exemple `System.in` qui référence les entrées clavier
- `String next()` retourne le prochain mot
- `int nextInt()` retourne le prochain mot, interprété comme un entier

Utilisation de librairies Java - Exemple de Scanner

Avec une sélection de méthodes publiques :

- `Scanner(InputStream source)` : constructeur
qui peut prendre en argument par exemple `System.in` qui référence les entrées clavier
- `String next()` retourne le prochain mot
- `int nextInt()` retourne le prochain mot, interprété comme un entier

fichier : TestScanner.java

```
import java.util.Scanner; // il faut importer la classe
public class TestScanner{
    public static void main(String[] args){
        Scanner s = new Scanner(System.in); // construction de cet objet
        System.out.println("entrez quelque chose"); // sachant que l'espace
            est la limite...
        String mot = s.next();
        System.out.println("Vous avez entré : " + mot);
    }
}
```

La semaine prochaine : exercices de modélisations

Extrait du Partiel 2018

- Un binôme est un objet qui se caractérise par une paire de String
- Ils sont numérotés de façon unique et automatique à leur création
- Leur représentation est normalisée : chaque paire a toujours sa première chaîne plus petite ou égale dans l'ordre lexicographique que sa seconde chaîne
- La constitution de ces binômes pourra changer avec le temps.
- On souhaite conserver un représentant des plus petits binômes qui ont déjà été créés (au sens lexicographique)

Écrire une classe Binome en justifiant vos choix.

(Écrivez aussi des méthodes auxiliaires utiles.)