Programmation orientée objet

Jeudi 8 septembre- vendredi 9 septembre

Des classes déjà vues

- Integer (contrairement à int, version classe des types classiques), Random, Scanner…

- On peut créer ses propres classes

- Définition :

Une classe est une entité qui définie un nouveau type

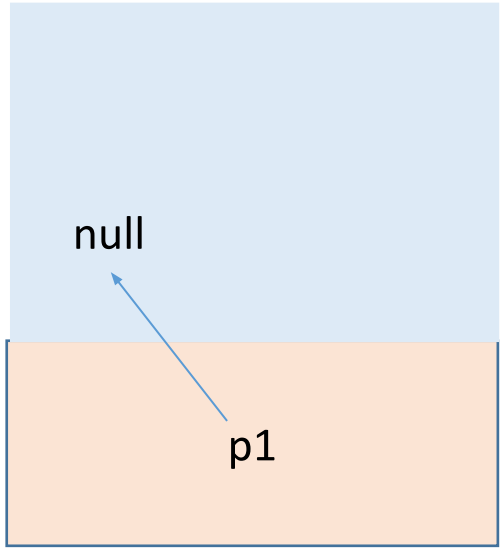
Exemple : classe Personne, qui permet de définir des personnes avec un nom, et prénom, un âge.
Peut dans un premier temps s'assimiler à des structures.

Une personne a certains type d'info, appelés attribut.

On va pouvoir définir des personnes particulières, ie, des instances

Comment se code une classe ?

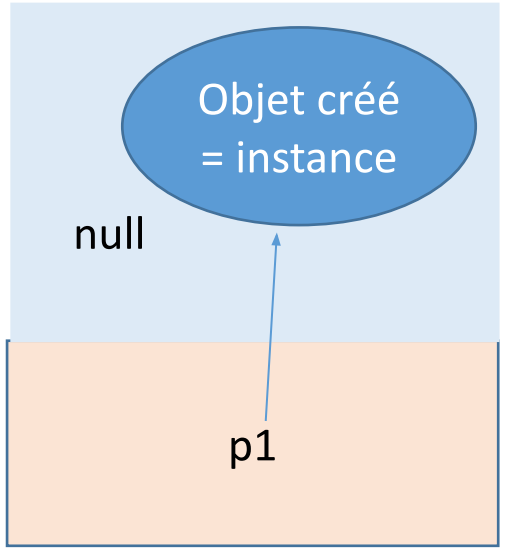
```
String str1; // declaration
str1="toto"; //affectation

Personne p1; //declaration : p1 de type Personne, une classe que l'on aura définie
// etape 1. Idem à Personne p1 = null
p1=new Personne(); //instanciation etape 2
p1=null; // retour au même état qu'apres etape 1, G.C. supprime l'objet (pas tjrs immédiat)
```



Heap managé

stack

étape1



Étape 2

Heap managé — Le garbage collector y supprime les instances vers lequelles ne pointent aucune référence

stack — Les objets s'y dépile « tout seul » à la fin du programme/méthode concerné

Vocabulaire

- p1 est une référence vers l'objet créé
- Classe utilisateur / classe métier (= nos classes qui tournet de manière sous-jacentes)

Créer ses classes : 1er exemple guidé

- Créer une nouvelle classe (sous eclipse) sans cocher main, donner un nom commençant par une majuscule (Personne) (classe métier) :

On a déjà une classe vide utilisable même si elle est vide

- Créer une classe TestPersonne contenant un main (classe de test)

Contenu de TestPersonne.java :

```
package debut;

public class TestPersonne {

public static void main(String[] args) {

Personne p= new Personne();//declarat. puis instanciat.
p.nom="toto";
p.prenom="titi";
p.age=10; //initialisation

System.out.println(p.nom+" "+p.prenom+" "+p.age);

}
}
```

Ne pas hésiter à mettre les différents tests dans des méthodes à part pour mieux organiser son main !

Contenu de Personne.java :

```
package debut;

public class Personne {
String nom;
String prenom;
int age;
}
```

Explication des contenus des classes

Que trouve-t-on dans une classe ?

- Attributs (aussi appelés data membre ou variables de classe ou d'instance – précision à venir) à différencier d'une variable locale (d'une méthode par exemple) : a une valeur par défaut
- Méthodes
- Constructeur

```
package debut;

public class Personne {
String nom;
String prenom;
int age;
}
```

attributs

```
Personne p1= new Personne();
p1.nom="toto";
p1.prenom="titi";
p1.age=10;

Personne p2= new Personne();
p2.nom="aa";
p2.prenom="bb";
p2.age=20;
```

Deux instances
Complètement indépendantes

Des erreurs à connaître

```
p1=null;
System.out.println(p1.nom+" "+p1.prenom+" "+p1.age);
```

Déclenche un erreur d'exécution (compile bien) :

```
Exception in thread "main" java.lang.NullPointerException
at debut.TestPersonne.testinstanciation(TestPersonne.java:22)
at debut.TestPersonne.main(TestPersonne.java:6)
```

Signifie que l'on tente d'utiliser un objet non-instancié :
On ne peut pas y accéder !

Des valeurs par défaut

```
Personne p1= new Personne();
System.out.println(p1.nom+" "+p1.prenom+" "+p1.age);
```

On obtient à l'exécution l'affiche :

null null 0

Valeur par défaut de int

Valeur pas défaut pour String

Lors de l'instanciation, les attributs prennent leur valeurs par défaut

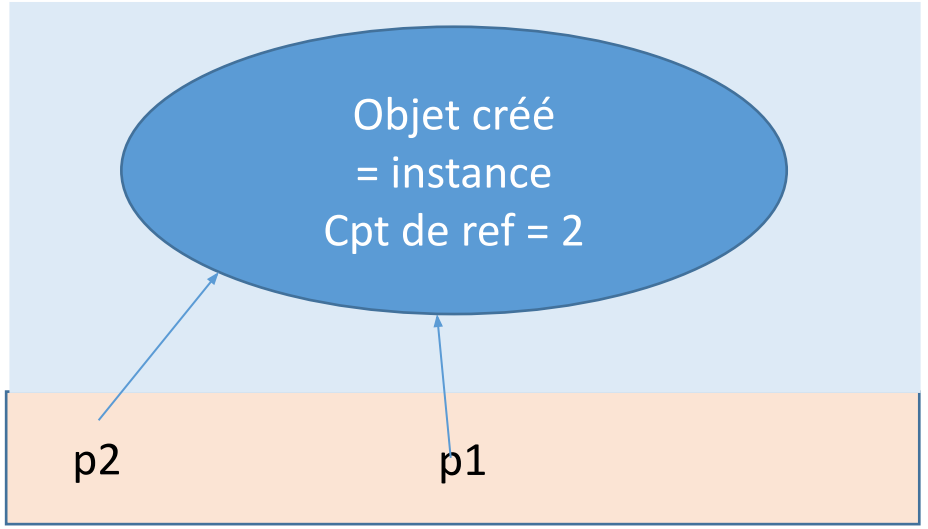
Références

```
Personne p1= new Personne();
p1.nom="toto";
p1.prenom="titi";
p1.age=10;
Personne p2= p1; // crée une deuxième ref vers le même objet
p1=null; // l'instance n'est pas supprimé : il reste encore une référence dessus
p1=p2; // p1 repointe vers l'instance
System.out.println(p1.nom+" "+p1.prenom+" "+p1.age);
System.out.println(p2.nom+" "+p2.prenom+" "+p2.age);
```

Fonctionne !

Attention : Ca ne crée pas de clone de l'instance !

Schéma mémoire après `Personne p2= p1;` :



Une méthode dans notre classe

```
package debut;

public class Personne {
String nom;
String prenom;
int age;

void affiche(){
System.out.println(nom+" "+prenom+" "+age);
}

}
```

Pas de static dans la classe métier (schématiquement)
Sinon même manière de déclarer

La déclencher dans la classe test :

```
p1.affiche();
```

Référence vers l'instance

Nom de la méthode dans la classe

Constructeurs

```
Personne p;
p= new Personne();
```

Constructeur : deux rôles :

- Crée l'objet en mémoire
- Affecte les valeurs par défaut

On a pourtant pas écrit ce constructeur. C'est parce qu'il existe différents types de constructeurs.

1. Constructeur par défaut de la JVM (= constructeur implicite) : si aucun constructeur n'est codé, qui permet de créé un objet et donne aux attributs la valeur par défaut de leur type, on a donc une instance utilisable.
   C'est ce qui se passe dans l'exemple ci-dessus.
   N'existe que s'il n'y a aucun constructeur dans la classe
2. Constr. par défaut : ex. par slide suivant
3. Constr. d'initialisation : lorsqu'il y a au moins un paramètre en entrée
4. Constr. par copie

Un constructeur par défaut

- Un constructeur porte toujours le même nom que la classe (exactement, avec une majuscule)
- Ressemble à une méthode, mais n'a aucun type retour
- Un constructeur **par défaut** ne prend pas de paramètre

Exemple :

Dans la classe Personne :

```
Personne(){
}
```

Dans classe de test :

```
Personne p;
p= new Personne();
```

C'est maintenant notre constructeur qui est appelé (ici fait exactement la même chose)

Autre version :

```
Personne(){
nom="Dupont";
}
```

Donnera la valeur « Dupont » à nom lorsque l'on appellera notre constructeur (en écrasant le null qui arrive automatiquement à la cration de l'objet

Etape d'une instanciation

1. Création + mettre aux valeurs par défaut
2. Initializeur (remplace les valeurs par défauts) par celles définies au niveau des attributs (même dans le cas du constr. JVM), les valeurs ne peuvent être que des constantes.
3. Execution du contenu du constructeur (dans lequel on peut faire un traitement)
4. Affectation de l'extérieur depuis la référence

```
public class Personne {
String nom = "Defaut";
String prenom;
int age;

Personne(){
nom="Dupont";
}
}
```

Appelé dans la classe test :

Personne p1= **new Personne();**

p1.nom="toto";

Valeurs prises par p1.nom selon les étapes :

1. Null
2. « Defaut »
3. « Dupont »
4. « toto »

Au moment du **new Personne();**

Différence constructeur/méthode

- Le constructeur n'est appelé qu'une seule fois lors de la vie d'une instance : lors de sa création.
- Une méthode peut être appelé à différents moment de la vie d'une instance, autant de fois que l'on veut.

Exemple :

```
Personne p = new Personne();
p.traitement();
p = new Personne(); // crée une nouvelle instance et change l'objet
//auquel p fait référence. L'instance précédent n'étant plus référencée
//elle sera supprimée par le G.C.
```

Instance anonyme

- Exemple :
  new Personne();

  instance sans référence qui pointe dessus, donc sont compteur à 0, sera supprimée par le G.C.

Constructeur d'initialisation

```
Personne(String n){
nom=n;
}
```

Au moins un paramètre : constructeur d'initialisation

Si l'on a que des constructeurs d'initialisation, le constructeur implicite ne peut plus se déclencher, Personne(); ne marche pas si pas défini.
Les constructeurs peuvent se surcharger.

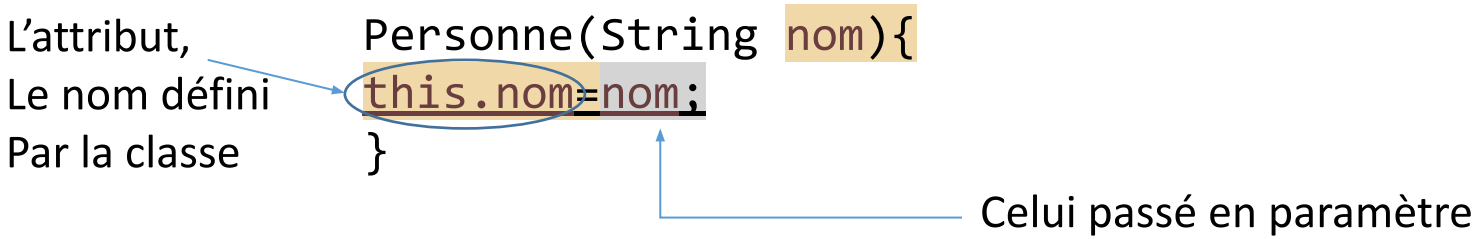
A appeler comme ça :
Personne p1= **new Personne("Moi");**

On peut en avoir plusieurs différents pour renseigner tout ou partie des attributs, mais on est limité par le fait qu'il faut avoir à chaque fois des paramètres d'entrées différents, comme pour les surchages de méthodes

Consulter les constructeurs dispo

Apres un new Personne ctrl+space premet de voir la liste des constructeurs

- Paramètres p, a, n etc pas très parlant, on voudrait pouvoir donner des noms parlants comme prenom, age, nom.
- Risque de confusion avec les noms d'attributs, dont on voudrait utiliser exactement les mêmes noms pour savoir ce que l'on fait : une solution simple le « this »



Programmation Orientée Objet

- But : réutilisation sous toutes ses formes
- Un constructeur peut faire appel à un autre (si un bout du traitement est commun, pour pas avoir à le refaire

```
Personne(String nom, String prenom){
this.nom=nom;
this.prenom=prenom;
}

Personne(String nom, String prenom, int age){
this(nom,prenom);
this.age=age;
}
```

Appelle le constructeur de la classe avec les paramètre nom et prenom

Limite : on ne peut appeler qu'un seul constructeur (qu'un this) et forcément en première position

Variante de nommage

- Certains package utilise des attributs préfixé de _
  (par exemple int _age) pour éviter les this (dans des cas plus ou moins restreints et normés)
- Important : garder des noms de variables clairs pour l'utilisateur

Constructeur par copie

- Permet à l'utilisateur de cloner un objet :
  on a une instance en mémoire, que l'on veut conserver, et en créé une autre identique mais que l'on pourra manipuler sans modifier la première (instance indépendante)

```
Personne(Personne x){
nom=x.nom;
prenom=x.prenom;
age=x.age;
}
```

Prend en paramètre une instance

Met les attributs à la même valeur que dans l'instance donnée

Tableau de personnes : déclaration

```
Personne[] tab = new Personne[3];
```

Pas un constructeur de Personne !

Crée un tableau de 3 personnes, chacune étant null pour l'instant

```
static void test2(){
Personne[] tab = new Personne[3];//cree un tableaude Personne de taille 3
Personne p1=new Personne("A", "B", 41); //instancie un personne dont p1 est une ref.
tab[0]=p1; //tab[0] est une nouvelle référence vers l'instance cree au-dessus
p1.affiche();
tab[0].affiche(); //idem à la ligne precedente
p1=null; // l'instance a toujours un pointeur vers lui
tab[1]= new Personne("C", "D",20);//sans passer par une autre référence
tab[2]= new Personne("E","F",30);
}
```

```
for (Personne p : tab){
if (p!=null) p.affiche();
}
```

On aurait bien sur pû faire une boucle for plutôt que for each

Permet de boucler sur tout un tableau de personne pour effectuer un traitement voulu (ici, affiche) sans planter si le tableau est « trop grand », ie, si certaines cases ne sont pas des ref vers des instances effectivement créée.

Autour de la méthode affiche

- On a toujours (norme), une méthode nous permettant de nous renseigner sur les attributs de la classe (comme affiche ici)
- Deux problèmes dans notre classe affiche :
  1. On a décidé de l'appeler affiche mais le nom n'est pas normalisé, on voudrait une norme commune à toute les classes pour pouvoir facilement savoir comment appelé ce genre de méthode sur tout type de classe
  2. on voudrait pouvoir l'utiliser dans tout contexte (ici, limité à un projet de type console, car on a utilisé System.out.println). On va ensuite éviter les println dans les classes métiers pour pouvoir être utilisé dans différents cadre.

La méthode toString

```
public String toString(){ //il faut exactement cette signature
return nom+prenom+age; // on pourrait avoir un autre format
}
```

- N'utilise pas de println : renvoie un String
- Nom normalisé, toute les classes ont une méthode toString
- Pas de norme sur le formatque l'on renvoie
- Générable automatique avec eclipse (source, generate toString). On peut faire des constructeurs idem (et d'autres)

Exemples d'utilisation dans la classe utilisateur (qui elle est dépendante du support) :

```
for (Personne p : tab)
if (p!=null) System.out.println(p.toString());
```

Appel implicite d'un toString dans `System.out.println :`

```
Personne p1=new Personne("A", "B", 41); System.out.println(p1);
```

Fonctionne ! (dans les autres cas il faut appeler explicitement la méthode)

Méthode create

- Scenario : un utilisateur préfère utiliser une méthode create plutôt que des constructeurs

```
Personne create(String nom, String prenom, int age){
return new Personne(nom, prenom, age);
}
```

A surcharger avec tous les types de constructeurs

Utilisation :

```
static void test2(){
    Personne pivot=new Personne(); //il faut une instance quelque part pour
//pouvoir utiliser des methodes
    Personne p1=pivot.create();
    Personne p2=pivot.create("blab", "hj");
    Personne p3=pivot.create("haha", "moi", 4);
}
```

Instance anonyme

```
static void test3(){
System.out.println(new Personne().getHello());
}
```

```
String getHello(){
return "hello";
}
```

One-shot : déclenche une méthode une fois.
On peut aussi mettre une ref dessus si on pense faire plusieurs getHello (par ex.) : mieux dans ce cas, ça évite de faire plein de création d'instance
Si par contre c'est très rare, mieux vaut une instance anonyme (et donc supprimée vite, traine pas en mémoire)

Visibilité (private, public, protected, default/package)

- Devant toute méthode, attribut, constructeur il faut mettre private ou public (on verra les autres cas plus tard)

Passons tout en public :

```
public class Personne {
public String nom = "Defaut";
public String prenom;
public int age;

public Personne(){
nom="Dupont";}

[…]

public String toString(){
return nom+prenom+age;
}

public String getHello(){
return "hello";
}
}
```

Que veut dire public/private ?

- Public : Accessible de l'extérieur de la Classe depuis une instance

  Exemple : p.methode(); dans une classe test : j'ai pu accédé à methode

- Private : non accessible depuis l'extérieur de la classe, la méthode n'est pas visible depuis d'autres classe, comme si elle n'existait pas.

  Une tentative d'utilisation d'une telle méthode depuis l'exterieur déclenche une erreur :

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
The constructor Personne(String, String, int) is not visible
```

- N'a de sens qu'à l'extérieur de la classe, dans une même classe toutes les méthodes ses voient les unes les autres.

```
private void m1(){
System.out.println("je suis m1");
}

public String getHello(){
m1();
return "hello";
}
```

On peut pas appeler directement m1 de l'extérieur
getHello peut faire appel à m1.
On peut appeler getHello de l'extérieur, qui n'a alors pas de soucis pour exécuter m1.

Quels cas d’utilisation ?

- Constructeurs public sauf exceptions (une bonne raison de bloquer un ou plusieurs constructeurs)
- Méthodes : ca dépend
  - Méthodes « centrales » publiques, celle comme point d’entrée nécessaire au traitrement demandées par l’utilisateur
  - Méthode auxilliaires existantes pour factoriser du code, utilisée juste par d’autres méthodes de la classe : private
- Attributs : private

  Pourquoi ? Sensible, mieux vaut y accéder via des méthodes pour pouvoir conditionner les modifications

Les autres ?

- Protected : lié à l'héritage, dans quelques jours
- default : la visibilité par défaut, ce qui se passe quand ne met rien (ce que l'on a fait jusqu'à présent. En java, ça correspond à la visibilité **package** : agit comme public dans un package et private dès qu'on en change

  A ne pas utiliser en général : l'utilisateur utilisera dans un package différents alors que le dev. Teste dans le même package. Problème lors de la livraison.

Accesseurs (getter et setter)

- Pour répondre au fait qu'avec des attributs private on veut quand même y accéder.

```
public int getAge() {
return age;
}

public void setAge(int age) {
this.age = age;
}
```

Utilisation :

```
Personne p0=new Personne();
p0.setAge(25);
System.out.println(p0.getAge());
```

- Générable automatiquement (via menu source d'eclipse)

On peut controler les données avant d'accepter la modification :

```
public void setAge(int age) {
if (age>=0&&age<160) this.age = age;
}
```

On peut utiliser les setters dans les constructeurs pour garantir la cohérence des données :

```
private Personne(String nom, String prenom, int age){
this(nom,prenom);
setAge(age);
}
```