INFO0201

Introduction à la programmation orientée objet

Partie 1

Une classe, des objets…

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

Equipe pédagogique : J. Jonquet, Ch. Jaillet, P. Delisle

Plan

- Préambule
- Analyse d'un exemple simple
- COO : Conception Orienté Objet
- La vie des objets
- Contrôle d'accès
- Les méthodes
- Diagramme de classes
- Compléments techniques
- Conception d'un exemple complet

Préambule

- Info0101 : objectif = algorithmique
  - On « pense » traitement
    - ⇢ fonctions + procédures
  - Classe = moyen d'exécuter un code (tester un algo)
    - ⇢ une seule classe
    - ⇢ un seul main, qui appelle des fonctions/procédures
- Info0201 : objectif = données
  - On « pense » données
    - ⇢ les données caractérisant un élément
    - ⇢ lorsque les éléments sont structurés
  - Classe = description des données (de la structure d'une catégorie d'éléments)
    - ⇢ on décrit des éléments, on les utilise => 2 classes
    - ⇢ Il peut donc y avoir des traitements (bien sûr !) :

      des traitements sur les données

Analyse d'un exemple de classe

- Exemple : une carte bancaire

  Qu'est-ce qu'une carte bancaire ? Qu'est-ce qui la caractérise ?

  Pour commencer on va se contenter de :

  - Un propriétaire
  - Un numéro de carte
  - Un code secret
  - Un plafond de paiement

- Modèle

  Chaque caractéristique sera enregistrée comme un **attribut** de la classe avec un nom, un type, un niveau d'accès :

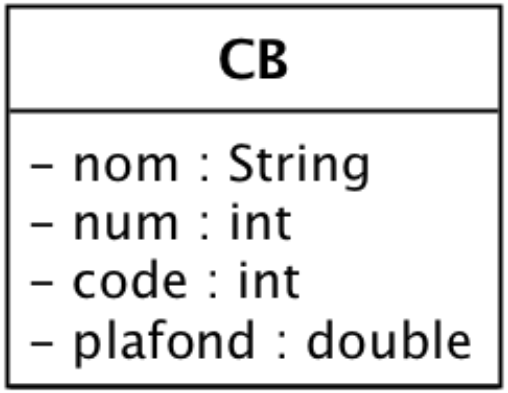
	Propriétaire	N° de carte	Code secret	Plafond de paiement
nom attribut	nom	num	code	plafond
type	ch. de carac.	entier	entier	réel
niveau d'accès	privé	privé	privé	privé

Analyse d'un exemple de classe

- Modélisation

	Propriétaire	N° de carte	Code secret	Plafond de paiement
nom attribut	nom	num	code	plafond
type	ch. de carac.	entier	entier	réel
niveau d'accès	privé	privé	privé	privé

Diagramme de classes



Code Java

```
class CB {
        //attributs
        private String nom;
        private int num;
        private int code;
        private double plafond;
}
```

Analyse d'un exemple de classe

- De la classe à l'objet

Nous avons caractérisé la carte bancaire, il est maintenant temps d'en créer une, voici ce que l'on souhaite :

Ma première CB

nom : "JONQUET"
num : 4978 7456
code : 1234
plafond : 300 €

Code Java

Par exemple dans une classe de test avec juste la procédure principale :

```
//declaration de la variable
CB premiereCB;

//instanciation de la classe
premiereCB = new CB("JONQUET", 49787456, 1234, 300.0)
```

Pour cela il faut un constructeur dans notre classe qui va créer l'objet !!

Analyse d'un exemple de classe

- De la classe à l'objet

On complète donc le code de notre classe :

Code Java

```
class CB {
        //attributs
        private String nom;
        private int num;
        private int code;
        private double plafond;

        //constructeur par initialisation
        public CB(String n, int nu, int c, double p){
                nom = n;
                num = nu;
                code = c;
                plafond = p;
        }
}
```

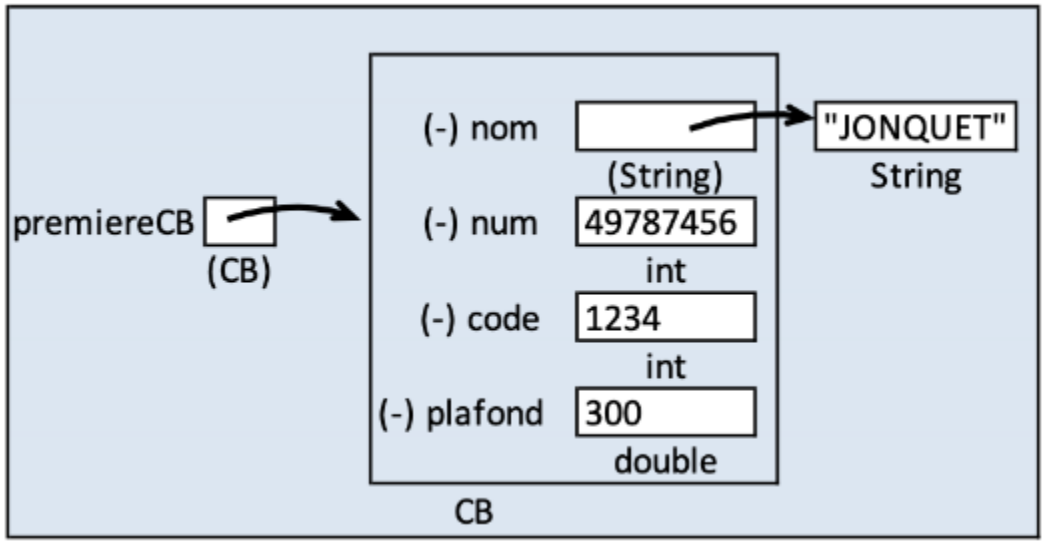
Accès direct aux attributs dans la classe

Permet d'affecter une valeur à chaque attribut de la classe pour l'objet que l'on crée

Analyse d'un exemple de classe

- De la classe à l'objet

Voyons la représentation mémoire :



L'**instanciation** permet de créer l'objet dans un espace réservé de la mémoire et renvoie sa **référence** qui sera stockée dans la **variable** `premiereCB` de **type** `CB`

On a créé notre première classe et notre premier objet mais dont l'utilité est, pour le moment, limitée… Ajoutons quelques fonctionnalités.

Analyse d'un exemple de classe

- Comportement du plafond de la CB

  Qu'est-ce que le plafond d'une carte bancaire ?

  Est-il consultable ? modifiable ? en dehors de la classe.

  Notre choix :

  - ✓ Consultable hors de la classe
  - ✓ Modifiable hors de la classe

Attribut privé => il faut donc ajouter des méthodes pour y accéder et le modifier

Code Java

```
…
// accès
public double getPlafond(){
        return plafond; // retourne la valeur de l'attribut plafond de l'objet courant
}
// modification
public void setPlafond(double p){
        plafond = p; // modification de la valeur de l'attribut de l'objet courant ici
}                                                    sans vérification…
```

Analyse d'un exemple de classe

- Et le code de la CB ?

  Est-il consultable ? Modifiable ? en dehors de la classe.

  Notre choix :

  - ✓Non-consultable en dehors de la classe
  - ✓Non-modifiable en dehors de la classe
  - ✓Mais il faut pouvoir le vérifier !

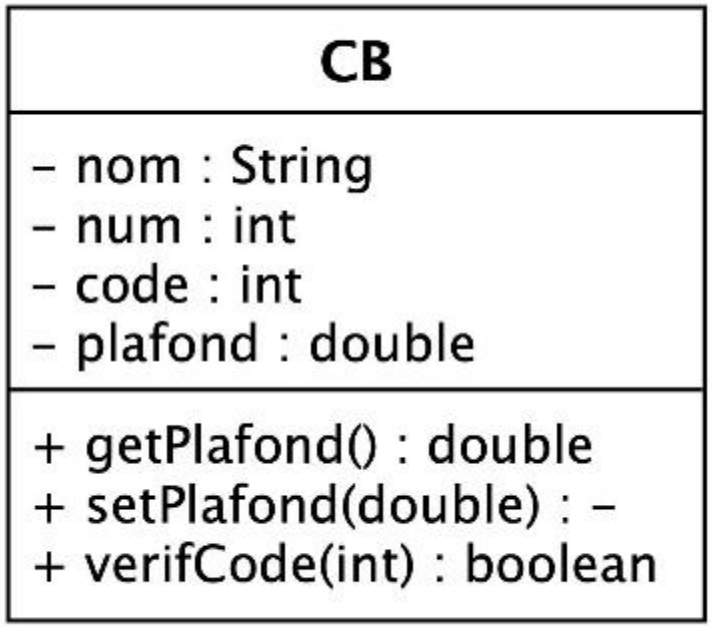
➔ Ajout d'une méthode verifCode : **Code Java**

```
…
// vérification du code
public boolean verifCode(int c){
        return (code == c); // retourne le résultat de la comparaison (vrai ou faux)
}
```

Et le diagramme de classes ?

Analyse d'un exemple de classe

- Nouveau diagramme de classes



➔ On ne met pas les constructeurs

On peut maintenant faire une classe de test !

Analyse d'un exemple de classe

- Utilisation dans une autre classe **Code Java**

```
class TestCB { // juste la procédure principale (main)
    public static void main (String[ ] args) {

        // déclarations des variables
        CB carte1, carte2;

        // instanciations
        carte1 = new CB("JONQUET", 49787456, 1234, 300.0);
        carte2 = new CB("JAILLET", 48875540, 9998, 300.0 );

        //modification du plafond de carte1
        carte1.setPlafond(1000);

        //vérification du code de carte2
        if(carte2.verifCode(1234))  System.out.println("code bon");
        else                        System.out.println("code faux");
    }
}
```

Notation pointée
Accès par la référence

Analyse d'un exemple de classe

- Utilisation dans une autre classe **Compilation Java**
  - Classe CB ➔ fichier **CB.java**
  - Classe TestCB ➔ fichier **TestCB.java**
  - Compilation des deux fichiers

```
$> javac CB.java
$> javac TestCB.java
```

➔ CB.class

➔ TestCB.class

  - Exécution de la classe de test

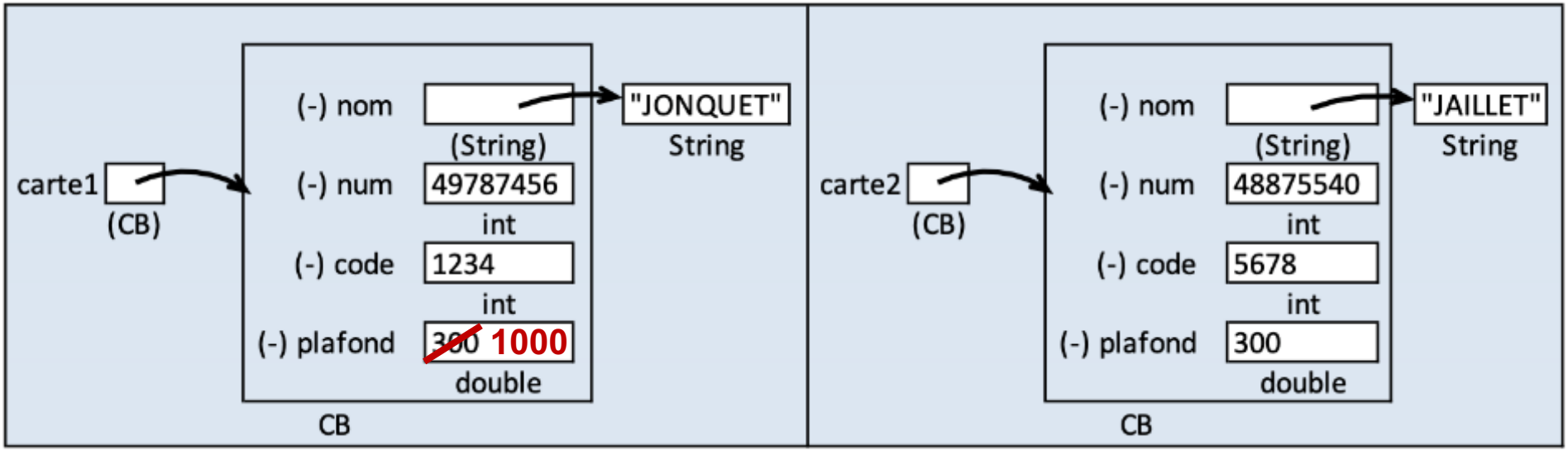
```
$> java TestCB
```

Attention les 2 fichiers doivent être dans le même dossier

Analyse d'un exemple de classe

- Utilisation dans une autre classe

Représentation mémoire



Plaçons bien les choses avant de poursuivre.

COO : Conception Orientée Objet

1. Analyser les différents types d'éléments en présence
2. Définir leurs caractéristiques (les données les caractérisant)
3. Expliciter l'ensemble des traitements qu'ils peuvent réaliser / subir

➔ **Programmation dirigée par les données**

- **TDA** : type de données abstrait
  - ensemble d'éléments muni d'opérations agissant sur ses éléments :
    - Données membres (éléments caractéristiques) : attributs
    - Opérations / traitements : méthodes
- **UML** : Unified Modeling Language
  - Moyen de modélisation => diagramme UML
- **Module** : entité regroupant l'ensemble des membres
  - Attributs et méthodes => encapsulation
- **Langage Orienté Objet**
  - Décrit les TDA
  - Organisation modulaire

COO : Conception Orientée Objet

- Terminologie
  - **Diagramme de classes (UML)** : représente les membres (attributs et méthodes) du modèle au sein du module et l'interaction entre les modules
  - **Classe** : modèle décrivant les caractéristiques communes et les comportements communs d'un ensemble d'éléments (module décrit selon le langage choisi)
  - **Objet** : représentant d'un classe
    - Une classe constitue un ***générateur*** d'objet / un ***modèle*** d'objet
    - Un objet est une ***instance*** de cette classe
  - **Membres** :
    - ***Attributs*** : données membres
    - ***Méthodes*** : comportement des objets de la classe

COO : Conception Orientée Objet

- Terminologie
  - **Instanciation**
    - Concrétisation d'une classe en un objet particulier
    - Code : utilisation de l'opérateur ***new*** avec un constructeur
  - **Caractérisation**
    - Un objet est caractérisé par les ***valeurs*** de ses attributs
    - Son comportement est défini par les méthodes de sa classe

- ***Classe*** : concept, description
- ***Objet*** : représentant concret d'une classe
- Une classe constitue un ***générateur*** d'objets
- Un objet est une ***instance*** de cette classe

La vie des objets

- Référence
  - **Les variables**
    - Pour un type primitif : contient sa valeur
      - Exemple : `int a = 5;`
    - Pour un tableau : contient une référence vers le tableau
      - Exemple : `int[] tab = new int[5];`
    - Pour un objet : contient **une référence** vers l'**objet**
      - Exemple : `CB carte1 = new CB("JONQUET",…);`

➔ **Attention au test d'égalité et à l'affichage des objets et tableaux**

- `S.o.p(a);` => ok
- `S.o.p(tab);` => ok mais `[I@4b1210ee`
- `S.o.p(carte1);` => ok mais CB@12a3a380

La vie des objets

- Référence
  - **Manipulation hors de la classe qui le définit**
    - Accès à un **attribut** si *public*
      ⇒ Par référence, notation pointée
    - Accès à une **méthode** si *public*
      ⇒ Idem, par référence, notation pointée
      ⇒ Doit être définie dans la classe décrivant l'objet
      ⇒ L'objet est un paramètre **implicite** de la méthode appelée
      La méthode est appelé sur l'objet donc connaît l'objet !!
    - Exemple : appel de la méthode `verifCode` sur l'objet `carte2` avec comme paramètre un `int` de valeur `1234`

```
//vérification du code de carte2
        if(carte2.verifCode(1234))   System.out.println("code bon");
        else                         System.out.println("code faux");
```

La vie des objets

- Référence
  - **Manipulation dans la classe qui le définit**
    - Accès à un **attribut** (privé ou public)

      ⇒ Accès direct sans notation pointée
    - Accès à une **méthode** (privé ou public)

      ⇒ Idem, accès direct sans notation pointée
    - Exemple : dans la méthode `verifCode` de la classe CB
      - code désigne l'attribut `code` de l'objet courant
      - la référence de l'objet courant est implicite

```
// vérification du code
    public boolean verifCode(int c){
        return (code == c); // retourne le résultat de la comparaison (vrai ou faux)
    }
```

La vie des objets

- Construction d'un objet
  - **Constructeur**
    - Fixe les valeurs de l'ensemble des attributs d'un objet
    - Toute classe devrait en posséder **au moins un**
    - Il peut en exister plusieurs versions pour une même classe, en principe :
      - **Par initialisation**
        - ⇢ On précise toutes les valeurs de tous les attributs en paramètres
        - ⇢ Attention : on vérifie que les valeurs sont correctes
      - **Par défaut**
        - ⇢ On ne précise rien (aucun paramètre), les valeurs sont fixées par défaut, selon une convention
      - **Par copie**
        - ⇢ On passe un objet de même type en paramètre, le nouvel objet aura les mêmes valeurs que celui passé en paramètre

    On peut en ajouter d'autres avec seulement une partie des attributs, selon les besoins.

La vie des objets

- Construction d'un objet
  - **Constructeur**
    - C'est une pseudo méthode
    - Sans type de retour
    - Son nom est celui de la classe
  - **Construire**
    - Utilisation de l'opérateur **`new`** : opérateur d'instanciation
  - **Allocation dynamique**
    - Déclenchement automatique du bon constructeur
      - Se fait en fonction des paramètres passés lors de l'appel : leurs nombres et leurs types
      - Mécanisme de lookup
    - Allocation à l'exécution

La vie des objets

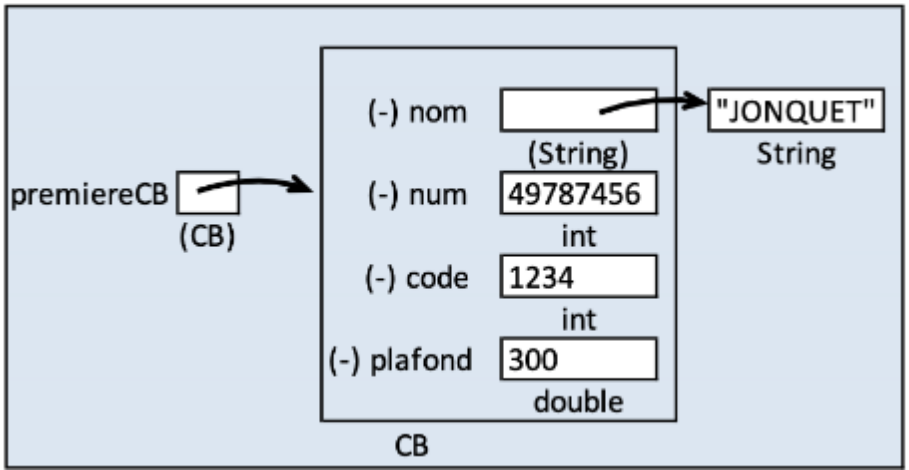
- Appel d'un constructeur
  - **Exemple**

```
//1. declaration de la variable
CB premiereCB;
//2. instanciation de la classe
premiereCB = new CB("JONQUET", 49787456, 1234, 300.0)
```

Analyse :

1. premiereCB variable de type CB, non initialisée
2. Réservation en mémoire et initialisation des attributs
   Affectation à la variable premiereCB de la référence de cet objet

Représentation mémoire :



La vie des objets

- Les différents constructeurs
  - **Exemple**

```
class CB {…
    //constructeur par initialisation
    public CB(String n, int nu, int c, double p){
        nom = n;

        if(nu<100000000)
                num = nu;
        else    num = 0;

        if(c<=9999)
                code = c;
        else    code = 0;

        if(p>=0)
                plafond = p;
        else    plafond = 0;
    }
```

```
//constructeur par défaut
    public CB(){
        nom = "personne";
        num = 0;
        code = 0;
        plafond = 0;
    }
    //constructeur par copie
    public CB(CB c){
        nom = c.nom;
        num = c.num;
        code = c.code;
        plafond = c.plafond;
    }
…
```

La vie des objets

- Suppression d'un objet
  - **Uniquement lorsqu'il n'est plus référencé**
    - C'est-à-dire lorsque plus aucune variable ne possède la référence pour l'atteindre
    - Exemple : mettre la variable à `null`

      ➔ attention si une seule variable pointe sur lui

  - **Ramasse-miette (garbage-collector)**
    - Thread de faible priorité : en tâche de fond, pas instantané
    - Libère l'espace mémoire des objets non-référencés
    - Possibilité de forcer l'appel : `System.gc();`

Contrôle d'accès

- Violation d'accès
  - Eviter tout accès (si non-autorisé)
    - Exemple : l'attribut `code` d'un objet CB
  - Eviter toute modification (si non-autorisé)
    - Exemple : le nom du propriétaire de la CB
  - Eviter les valeurs incohérentes
    - Exemple : mettre le plafond de la CB à -500…
    - Permet d'ajouter des vérifications, ici > 0

Contrôle d'accès

- Protection
  - Empêcher l'accès direct ➔ **contrôle d'accès**

Sécurité = encapsulation + contrôle d'accès

  - **Comment ?**
    - Interdire l'accès direct aux attributs
    - Modification uniquement par des méthodes (si autorisé)

      ➔ Permet de faire des vérifications (y compris dans un constructeur !)

  - **Application**
    - Interdire l'accès / l'utilisation ➔ privé
    - Autoriser l'accès / l'utilisation ➔ public
    - Doit être défini pour tous les membres (attributs et méthodes)

Contrôle d'accès

- Protection
  - **Modificateurs d'accès**
    - **public**
      - Accès inconditionnel
      - Depuis tout emplacement
    - **private**
      - Accès limité
      - Uniquement dans la classe où le membre est déclaré
    - Il y en a d'autres, on les découvrira en temps voulu
  - **Vocabulaire**
    - Partie visible = *public* = **Interface** (attributs publics et signature des méthodes publiques)
    - Partie cachée = *private* = **Implémentation** (attributs privés, méthodes privées et code des méthodes publiques)

Contrôle d'accès

- Protection

  - **Règles de bonne conduite**
    - Attribut : privé
    - Méthode : public
    - Routine : privé

      ⇒ Routine : méthode utilitaire, non accessible à l'utilisateur

  - **Schématisation dans les diagrammes de classes**
    - Public : +
    - Privé : -

Méthodes

- Généralités
  - **Fonctions-procédures / méthodes**
    - Regroupement d'un ensemble d'instructions qui constituent un traitement générique
    - On peut leur passer des paramètres
    - Elles peuvent renvoyer une valeur ou non
  - **Appel d'une méthode**
    - Message qu'on envoie à un objet ou une classe
    - Exemple :

```
carte1.setPlafond(500);
x = Math.sqrt(2);
```

NB: une fonction/procédure "classique" (INFO0101) est une méthode qui s'applique à une classe (la classe courante)

Méthodes

- Généralités
  - **Fonctions-procédures OU méthodes**

    - 2 cas ➔ 2 lieux
      - Traitement ponctuel ➔ dans la classe de test
      - Traitement plus courant ➔ dans la classe définissant l'objet

    - Appel :
      - Traitement réalisé par la classe
        ⇒ L'objet est passé en paramètre (une référence de l'objet)
      - Traitement réalisé par l'objet lui-même
        ⇒ L'objet est un paramètre implicite

Méthodes

- Généralités
  - **Fonctions-procédures OU méthodes : exemple**
    - On modélise des villes avec plusieurs caractéristiques dont les coordonnées GPS du centre de la ville.
    - On a besoin pour un traitement de calculer la distance entre 2 villes.

      ➔ Est-ce que nous en aurons besoin qu'une seule fois ? Est-ce que d'autres aussi pourraient s'en servir ?

Ponctuel :
fonction
dans la classe de traitement

```
// dans la classe de test
public static double calculDistance (Ville V1, Ville V2){…}
//appel dans le main de la classe de test
Ville reims = new Ville(…); //idem paris
double x = calculDistance(reims, paris);
```

Plus courant :
méthode
dans la classe définissant l'objet

```
// dans la classe Ville
public double calculDistance (Ville V){…}
//appel dans le main de la classe de test
Ville reims = new Ville(…); //idem paris
double x = reims.calculDistance(paris);
```

Méthodes

- Généralités
  - **Passage de paramètre**
    - **Par valeur**
    - Un appel de méthode **ne peut pas** modifier la valeur d'une variable passée en paramètre

1. Paramètres formels de type primitif

   ➔ aucune modification possible

```
public static void echanger (int a, int b){
      int tmp = a;
      a = b;
      b = tmp;
} // => ne fait rien
```

Les traitements réalisés sur les variables locales et les paramètres formels n'agissent pas sur les variables du main car **les paramètres effectifs sont des valeurs**.

Méthodes

- Généralités
  - **Passage de paramètre**
    - **Par valeur**
    - Un appel de méthode **ne peut pas** modifier la valeur d'une variable passée en paramètre

1. Paramètres formels de type primitif
2. Paramètres formels de type tableau ou objet
   - La valeur de la variable est la référence => non modifiable
   - Par contre l'objet ou le tableau référencé peut-être modifié
   - Valable aussi pour le paramètre implicite d'un appel de méthode (l'objet sur lequel est appelé la méthode)

```
carte1.setPlafond(500);
// modifie l'attribut plafond de l'objet référencé par carte1
```

`carte1` la référence : non modifié c'est **la valeur du paramètre effectif**

Méthodes

- Les méthodes classiques

1. **Accesseurs**
   - Méthodes de manipulation des attributs
   - Attribut par attribut
   - Permet d'éviter que l'utilisateur accède directement aux attributs
     - ⇢ indispensable dès que les attributs sont rendus privés
     - ⇢ sauf si on ne veut pas permettre de le consulter / modifier

   - Consultation
     - **getter** (du verbe anglais *get* => obtenir / récupérer)
     - Consultation de la valeur d'un attribut (toujours sauf cas particulier)

   - Modification
     - **setter** (du verbe anglais *set* => fixer)
     - Modification de la valeur d'un attribut (si on le permet)

Méthodes

- Les méthodes classiques

1. Accesseurs

➢ getter

- Indispensable pour consulter un attribut s'il est privé
- Nom : get + nom de l'attribut avec une majuscule
- Type de retour : celui de l'attribut
- Sans paramètre
- Public

Exemple : consulter le plafond d'une CB => attribut privé sans accès direct

- Appel (dans la classe de test)

```
double x;
x = carte1.plafond; // pas d'accès direct
x = carte1.getPlafond();
```

- Définition de la méthode (dans la classe personne)

```
public double getPlafond(){
        return plafond; // retourne la valeur de l'attribut plafond de l'objet courant
}
```

Méthodes

- Les méthodes classiques

1. Accesseurs

➢ setter

- Indispensable pour modifier si attribut privé et si modifications autorisées
- Nom : set + nom de l'attribut avec une majuscule
- Type de retour : aucun (`void`)
- Paramètre : un seul du type de l'attribut
- Permet de **vérifier** la nouvelle valeur avant de l'affecter
- Public

Exemple : modifier le plafond d'une CB => attribut privé sans accès direct

- Appel (dans la classe de test)

```
carte1.plafond = 1000; // pas d'accès direct
carte1.setPlafond(-300); // plafond non modifié
carte1.setPlafond(1000);
```

- Définition de la méthode (dans la classe personne)

```
public void setPlafond(double p){
        if(p >= 0)
            plafond = p; // fixer l'attribut plafond de l'objet courant si >= 0
}
```

Méthodes

- Les méthodes classiques

2. La méthode `toString()`

- Permet d'obtenir la description d'un objet sous forme d'une chaîne de caractères
- Prototype

```
public String toString() {…}
```

  - Méthode de transformation de l'état d'un objet en chaîne de caractères de type `String`
- Utilisation

```
CB carte1, carte2;
… // instanciation de carte1 et carte2
System.out.println ( carte1.toString() );
System.out.println ( carte2 );
```

  - Cette méthode est dite implicite : on peut donc utiliser directement la référence dans l'affichage et c'est cette méthode qui sera appelée.

    => si on ne la prévoit pas dans le code de la classe, elle est générée automatiquement par le compilateur et affiche la référence de l'objet

Méthodes

- Les méthodes classiques

3. La méthode `egalA()`

- Test d'égalité : compare l'objet courant (paramètre implicite) avec un autre objet passé en paramètre
- Type de retour : `boolean`
- Prototype

```
public boolean egalA(Type_de_la_classe ref) {…}
```

- Utilisation

```
CB carte1, carte2;
… // instanciation de carte1 et carte2
if( carte1.egalA(carte2) )
    …
```

- Il existe aussi la méthode `equals` qui compare l'objet courant avec un objet de type quelconque : plus technique, nous verrons cela plus loin

Méthodes

- Les méthodes classiques

Ajout de ces méthodes dans notre exemple

```
class CB {
        …
        //methode toString : description de l'objet sous forme d'une chaine de caracteres
        public String toString(){
                String s = "La carte bancaire n°"+num+" appartenant a "+nom+" a un
                plafond autorise de "+plafond+" euros.";
                return s;
        }
        //methode egalA : compare l'objet courant a celui passe en parametre
        //reflexion : qu'est-ce que 2 CB identiques ? Plusieurs possibilités
        public boolean egalA( CB ref ){
                return (num==ref.num && nom.equals(ref.nom) && code==ref.code &&
                plafond==ref.plafond);
        }
}
```

Diagramme de classes

- Pour une classe
  - NOM de la classe
  - Tous les attributs de la classe
    ⇒ avec leurs modificateurs d'accès (+ / -)
  - Toutes les méthodes
    ⇒ avec leurs modificateurs d'accès (+ / -)
    ⇒ attention pas les constructeurs

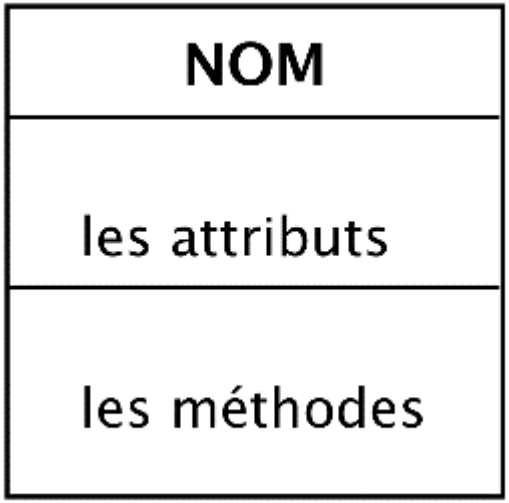
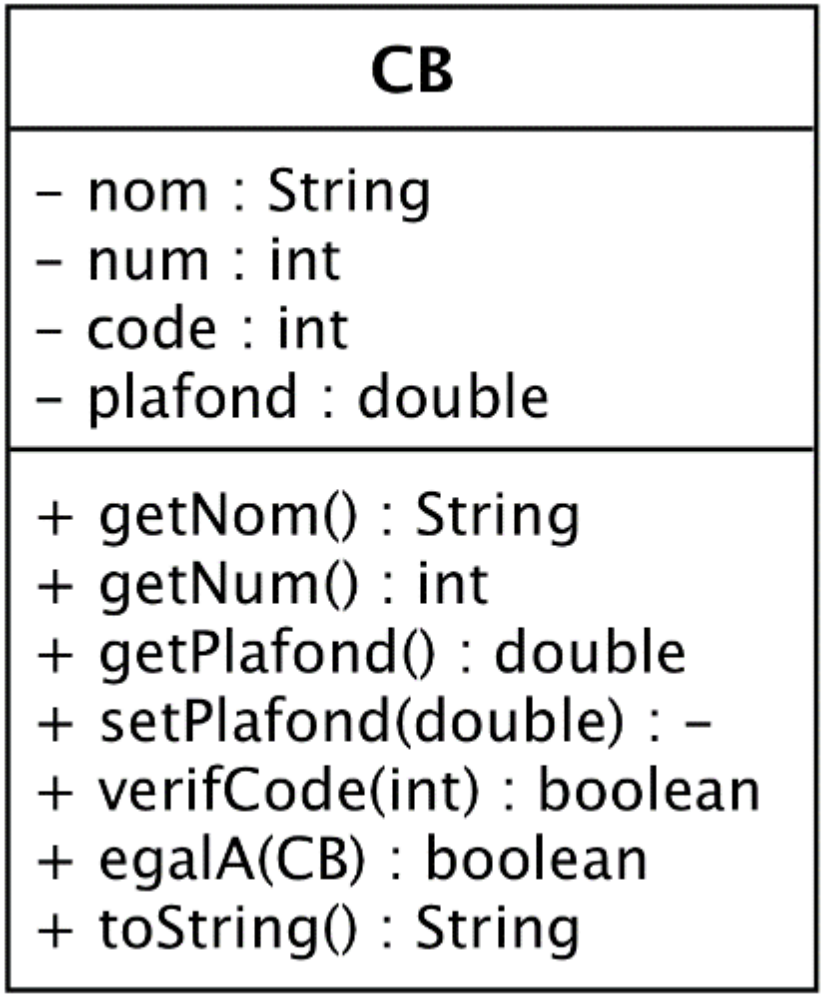


Diagramme de classes

- Notre classe exemple

  - Ajout des getters / setters manquants
  - Ajout des méthodes classiques



Compléments techniques

- Auto-référence : `this`
  - **Analyse du code d'une méthode**

```
//issue de la classe CB
    // vérification du code
    public boolean verifCode(int c){
        boolean ok = false;
        if ( code == c)
            ok = true;
        return ok; // retourne le résultat de la comparaison (vrai ou faux)
```

3 types de variables :

- **c** paramètre formel
  - La valeur est transmise depuis l'extérieur de la méthode
  - Dans le bloc, il joue le même rôle que les variables locales
- **ok** variable locale
- **code** attribut de l'objet courant

  => déclaration en dehors de la méthode

Compléments techniques

- Auto-référence : `this`
  - **`this` pour expliciter la référence de l'objet dans le code d'une méthode**

```
public double getPlafond(){
    return this.plafond; // retourne la valeur de l'attribut plafond de l'objet courant
}
```

Possible aussi pour les constructeurs

```
public CB(String n, int nu, int c, double p){
    this.nom = n;      this.num = nu;
    this.code = c;     this.plafond = p;
}
```

Synthèse

- Attention : obligatoirement dans la classe décrivant l'objet
- On définit les attributs (avec leur nom)
- Dans les constructeurs et méthodes, on les utilise
  - Soit directement => la référence de l'objet courant est implicite
  - Soit via l'auto-référence **`this`** => explicite et notation pointée

Compléments techniques

- Auto-référence : `this`
  - **`this` pour résoudre un masquage**

```
public void setPlafond(double plafond){
        if(plafond>=0)
                plafond = plafond; // ????
}
```

Résoudre le masquage de l'attribut par un paramètre ou une variable locale : l'expliciter

```
public void setPlafond(double plafond){
        if(plafond>=0)
                this.plafond = plafond;
}
```

Compléments techniques

- Auto-référence : `this`
  - **`this(…)` appel d'un autre constructeur**

  `this(…)` désigne un autre constructeur choisi en fonction des paramètres passés

  ➔ En pratique on part du constructeur par initialisation que l'on fait parfaitement avec toutes les vérifications et on l'appel dans les autres constructeurs

```
//constructeur par défaut
public CB(){
        nom = "personne";
        num = 0;
        code = 0;
        plafond = 0;
// ou
//      this("personne", 0, 0, 0);
}
```

```
//constructeur par copie
public CB(CB c){
        nom = c.nom;
        num = c.num;
        code = c.code;
        plafond = c.plafond;
// ou
//      this(c.nom, c.num, c.code, c.plafond);
}
```

Compléments techniques

- Surcharge
  - **Surchage = surdéfinition (*overloading*)**
    - Plusieurs méthodes de même nom dans une même classe
    - Type de retour identique
    - Elles diffèrent par leurs paramètres : nombres et/ou types
  - Choix de la version à appeler
    - Selon le contexte = nombre et types des paramètres
    - ***static lookup*** : choix statique (à la compilation)
  - Exemples :
    - Les constructeurs (par initialisation / par défaut / par copie)
    - Autre exemple :

```
public boolean verifCode(int c){…} // un entier inférieur ou égal à 9999
public boolean verifCode(int a, int b, int c, int d){…} // 4 chiffres entre 0 et 9
```

Compléments techniques

- Surcharge
  - Une version peut en appeler une autre
    - Exemple : les constructeurs
    - Autre exemple :

```
public boolean verifCode(int c){ // un entier inférieur ou égal à 9999
        return (code == c);
}
public boolean verifCode(int a, int b, int c, int d){ // 4 chiffres entre 0 et 9
        int codeTest;
        boolean ok = false;
        if( a >= 0 && a < 10 && b >= 0 && b < 10 && c >= 0 && c < 10 &&
                d>=0 && d<10){
                codeTest = 1000 * a + 100 * b + 10 * c + d ;
                ok = (code == codeTest ) ;
        // ou
        // ok = verifCode( 1000 * a + 100 * b + 10 * c + d );
        return ok;
}
```

Conception exemple complet

- Les personnes

Les questions à se poser pour établir le diagramme de classes et le code de la classe

- **Informations définissant une personne ?**
- **Données caractérisant une `Personne`**
- **Constructeur par initialisation : quelles vérifications ?**
- **Constructeur par défaut : quelles valeurs par défauts ?**
- **Constructeur par copie : est-il judicieux ?**
- **Getters : pour tous les attributs ?**
- **Setters : pour tous les attributs ?**
- **toString : quelles données affichent-on et comment ?**
- **egalA : comment compare-t-on 2 `Personnes` ?**
- **Méthodes supplémentaires**

Conception exemple complet

- Les personnes
  - **Informations définissant une personne ?**

Nom, prénom, sexe, date de naissance, âge, taille, couleur de peau, couleur de cheveux, langue parlée (plusieurs peut-être même), numéro de sécu, adresse, tatouages, …

=> Tout dépend de ce que l'on veut en faire

  - **Données caractérisant une `Personne`**

Selon ce qu'on souhaite faire, il faut choisir les données que nous considérons comme caractérisant notre classe `Personne`, ici :

- nom : String
- prenom : String
- age : int
- adresse : String

=> Les attributs de la classe `Personne`

Conception exemple complet

- Les personnes
  - **Constructeur par initialisation : quelles vérifications ?**

Nom, prénom, adresse : limite de taille ? Non pas spécialement

Age : positif ! (>=0 et <150 ?)

  - **Constructeur par défaut : quelles valeurs par défauts ?**

- nom : "X"
- prenom : "x"
- age : 0
- adresse : "Lune"

  - **Constructeur par copie : est-il judicieux ?**

Peut-on faire la copie d'une personne ? => non

Et finalement le constructeur par défaut ? …

Conception exemple complet

- Les personnes
  - **Getters : pour tous les attributs ?**

  Oui

  Et si on avait modélisé les tatouages ?

  - **Setters : pour tous les attributs ?**

    - `nom` : peut-on changer de nom ? Oui (mariage, …)
    - `prenom` : peut-on changer de prénom ? Oui (intérêt légitime)
    - `age` : peut-on changer d'âge ? Non mais on vieillit !
    - `adresse` : peut-on changer d'adresse ? Oui (déménagement, …)

Conception exemple complet

- Les personnes
  - **toString : quelles données affiche-t-on et comment ?**

  Ex : Christophe Jaillet a ?? ans et vit à Reims.

  - **egalA : comment compare-t-on deux `Personne` ?**

  Mêmes nom, prénom, âge et adresse ? Mais même comme cela sont-elles vraiment identiques ?

  Même âge est-il suffisant ?

  => Tout dépend du contexte : ici je choisis arbitrairement même âge et même adresse !

Conception exemple complet

- Les personnes
  - **Méthodes supplémentaires**

Tout dépend de ce que l'on a besoin de faire comme traitement.

On a vu qu'on peut vieillir :

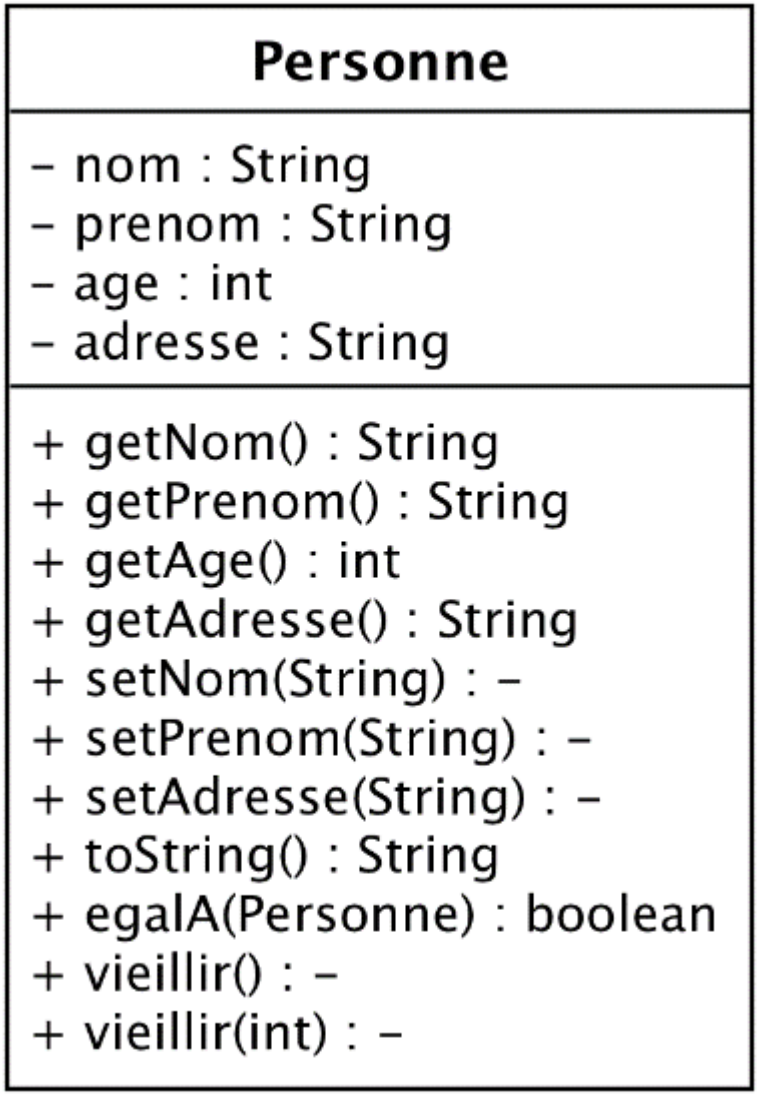
- année par année ? oui
- de plusieurs années d'un coup ? Mise à jour ponctuelle, oui mais > 0

➔ **On peut maintenant établir le diagramme de classes**

Conception exemple complet

- Les personnes
  - **Diagramme de classes**

Et le code java ?



Conception exemple complet

- Les personnes
  - **Code java de la classe**

```
class Personne {
    //attributs
    private String nom;
    private String prenom;
    private int age;
    private String adresse;

    //constructeur par initialisation avec verification
    public Personne(String n, String p,
                    int a, String ad){
        nom = n;   prenom = p;
        adresse = ad;
        age = 0;
        if(a>=0)  age = a;
    }

    //constructeur par defaut
    public Personne(){
        this("X","x",0,"Lune");
    }
    //constructeur par copie : non

    //getters
    public String getNom(){return nom;}
    public String getPrenom()
        {return prenom;}
    public int getAge(){return age;}
    public String getAdresse()
        {return adresse;}
}
```

Conception exemple complet

- Les personnes
  - **Code java de la classe suite**

```
//setters
public void setNom(String nom)
        {this.nom = nom;}
public void setPrenom(String p)
        {prenom = p;}
public void setAdresse(String ad)
                {adresse = ad;}

//description
public String toString(){
        return prenom + " " + nom + "
        a " + age + " ans et vit a " +
        adresse + ".";
}
```

```
//egalite
public boolean egalA(Personne ref){
        return age == ref.age &&
            adresse.equals(ref.adresse);
}

//vieillir d'un an
public void vieillir(){
        age++;
}
//vieillir : mise a jour ponctuelle
public void vieillir(int a){
        if(a > 0 && age+a < 150)
        age = age + a;
}
}
```

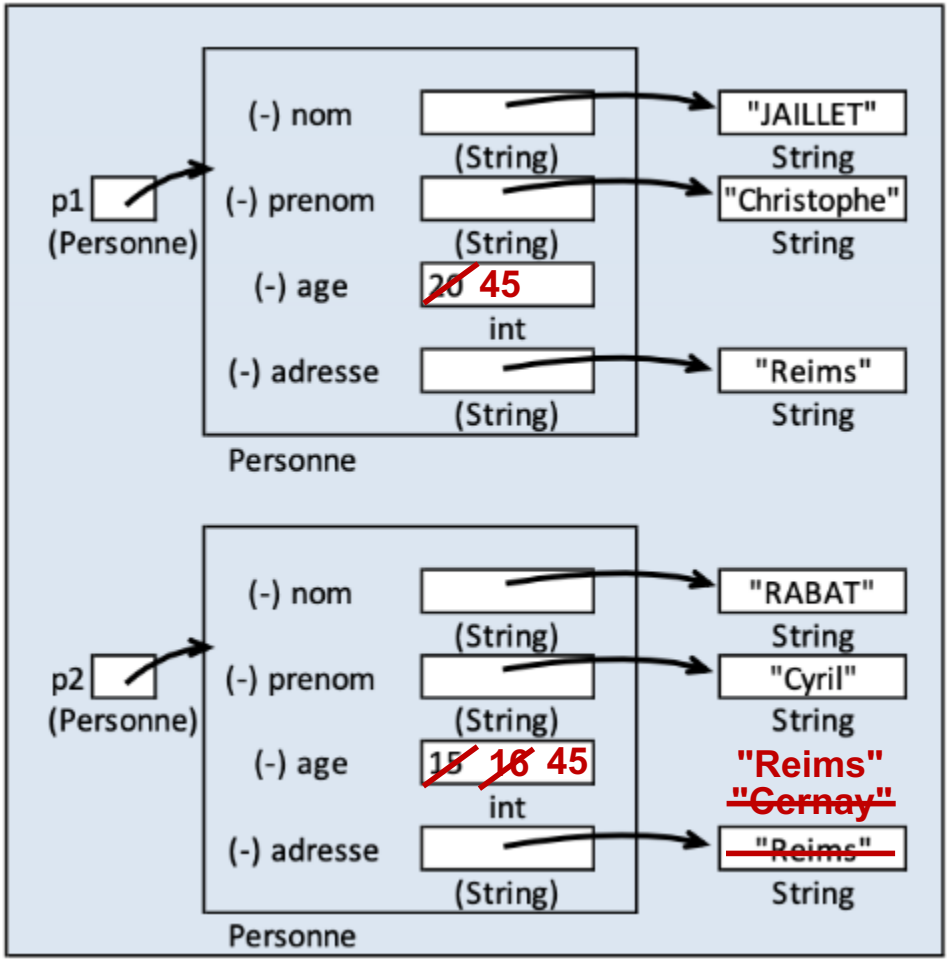
Conception exemple complet

- Les personnes **Et les traitements ? Code java de la classe de test**

```
class TestPersonne {
   public static void main (String[ ] args) {
      // déclarations des variables
      Personne p1 = new Personne("JAILLET", "Christophe", 20, "Reims");
      Personne p2 = new Personne("RABAT", "Cyril", 15, "Reims");
      //déménagement
      p2.setAdresse("Cernay");
      //modification des ages
      p1.vieillir(25);
      p2.vieillir();
      //affichage
      S.o.p(p1.toString());
      S.o.p(p2);
      //on modifie pour qu'ils soient identiques
      p2.vieillir( p1.getAge() - p2.getAge() ); p2.setAdresse( p1.getAdresse() );
      //comparaison
      if(p1.egalA(p2) != true)  S.o.p("C'est pas les mêmes !");
      else                S.o.p(p1.getNom() + " et " + p2.getNom() + "sont identiques !");
}
```

Conception exemple complet

- Les personnes
  - **Représentation mémoire et affichage**



Affichage

Christophe JAILLET a 45 ans et vit a Reims.

Cyril RABAT a 16 ans et vit a Cernay.

JAILLET et RABAT sont identiques !

Prochaine partie

Les membres d'instance / de classe

Interface et implémentation