Premier aperçu des classes
Instructions conditionnelles

Faculté des sciences, Université de Montpellier
UE HAI717I - Programmation par objets

Limite des types primitifs

Examen d'un programme manipulant des variables destinées à contenir des informations (numéro, solde, nom du client) sur deux comptes bancaires :

```
String numCpte1;
String num2;
String nomClientCpte1;
double soldeCpte1;
String nomClient2;
double s2;
```

Limite des types primitifs

Observations :

- en vert, informations sur le premier compte
- en bleu, informations sur le second compte
- éléments dispersés
- nommage variable, non standardisé, non uniforme, pas toujours clair
- risque d'erreurs, pas de garantie que les variables soient utilisées à bon escient

déclaration de variable	notion
String numCpte1 ;	numéro
String num2 ;	numéro
String nomClientCpte1 ;	nom du client
double soldeCpte1 ;	solde
String nomClient2 ;	nom du client
double s2 ;	solde

$\Longrightarrow$ Le concept de compte bancaire n'est pas mis en valeur

Limite des types primitifs

Suite du programme :

```
nomClientCpte1="Axel";
soldeCpte1=300;
numCpte1=null;
num2 = "XX34091";
nomClient2 = "Bérénice";
s2 = 500;
```

Limite des types primitifs

Des groupes d'instructions dans le code ne sont pas mis en valeur, par exemple, des affectations pour initialiser les différentes variables représentant les comptes

affectation de valeur	information et compte concerné
nomClientCpte1="Axel" ;	nom client compte 1
soldeCpte1=300 ;	solde compte 1
numCpte1=null ;	numéro compte 1
num2 = "XX34091" ;	numéro compte 2
nomClient2 = "Bérénice" ;	nom client compte 2
s2 = 500 ;	solde compte 2

Limite des types primitifs

Suite du programme :

```
System.out.println(nomClientCpte1+" possède le compte de numéro "
     +numCpte1);

System.out.println("num="+num2+" client="+nomClient2
     +" solde="+s2);
```

Limite des types primitifs

Des groupes d'instructions dans le code ne sont pas mis en valeur, par exemple, des affichages de données, qui peuvent ne pas être réalisés de manière systématique ni similaire

instruction d'affichage	compte
SOP(nomClientCpte1+" possède le compte de numéro "+numCpte1) ;	compte 1
SOP("num="+num2+" client="+nomClient2+" solde="+s2) ;	compte 2

SOP = System.out.println

Limite des types primitifs

Suite du programme :

```
soldeCpte1 = soldeCpte1*(1+0.01);
double taux = 0.05;
s2 = s2*(1+taux);
```

Limite des types primitifs

Des groupes d'instructions dans le code ne sont pas mis en valeur, par exemple, pour augmenter le solde en versant des intérêts. Ce n'est pas non plus réalisé de la même manière, par exemple, la formule pour le premier compte inclut la valeur du taux, la formule pour le second compte inclut une variable qui contient la valeur du taux.

instruction relative au versement d'intérêt	compte concerné
soldeCpte1 = soldeCpte1*(1+0.01) ;	compte 1
double taux = 0.05 ;	taux pour le compte 2
s2 = s2*(1+taux) ;	compte 2

Limite des types primitifs

Synthèse

- désorganisation
- présentations différentes
- difficulté pour comprendre le code peu structuré
- si on veut appliquer un schéma de comportement systématique et uniforme on doit recopier du code et être bien attentif en l'instanciant
- quand on voudra le changer, il faudra changer dans tous les instanciations
- les erreurs seront fréquentes
  - oublis ou incohérences dans les descriptions
  - oublis dans les initialisations
  - calculs effectués ou présentés différemment

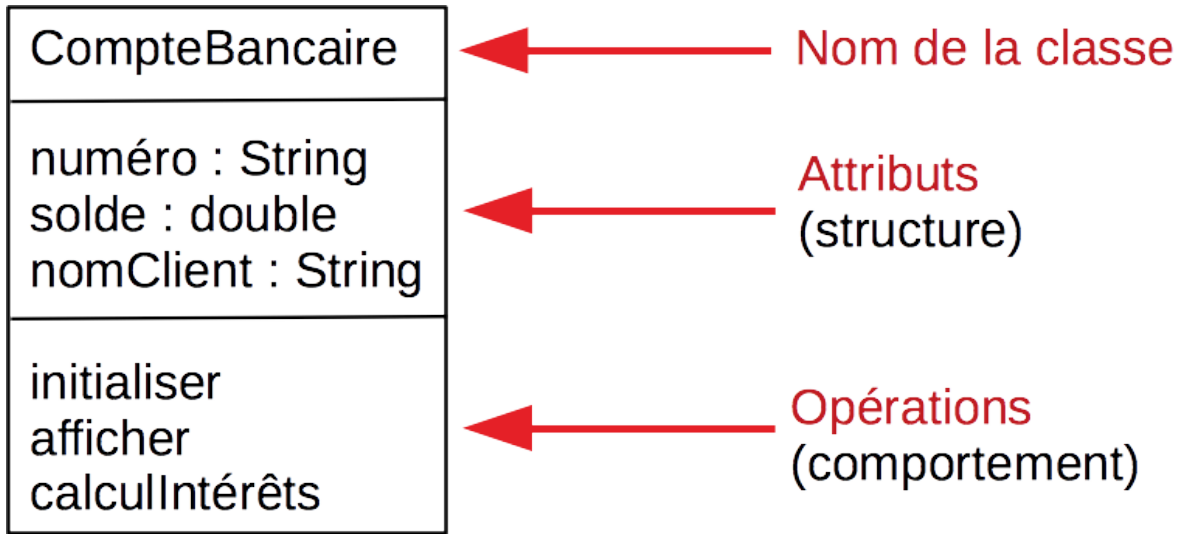
Introduire des représentations des concepts dans le code

Objets (instances) et classes

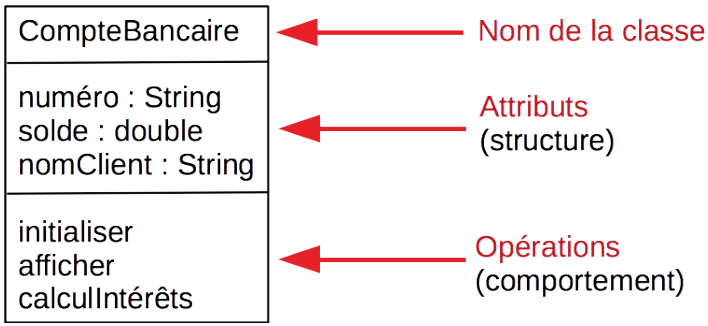
- dans le code on observe des **objets** (compte 1 et compte 2)
  - décrits par des données semblables (structure)
    - numéro
    - solde
    - nom du client
  - manipulés par des groupes d'instructions similaires (comportement)
    - initialiser
    - afficher des informations
    - verser des intérêts
- une **classe** `CompteBancaire` va être introduite pour représenter ces objets. Elle correspond à un **concept** du domaine de la banque.

Classe `CompteBancaire`

On peut la schématiser graphiquement par exemple ici avec le langage de modélisation UML



Classe CompteBancaire : structure en Java




```
public class CompteBancaire {
        private String numero;
        private double solde;
        private String nomClient;
...}
```

- la classe est `public` : la classe sera visible "partout"
- les attributs sont `private` : on ne pourra les manipuler que dans les opérations de la classe `CompteBancaire`

Création de comptes

On peut déjà :

- créer des objets dans un `main` grâce à l'opérateur new

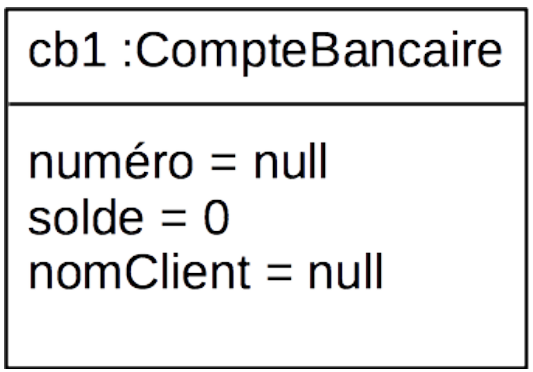
```
public class CompteBancaire {
  private String numero;
  private double solde;
  private String nomClient;

  public static void main(String[] a) {

    CompteBancaire cb1 = new CompteBancaire();  // creation
    ....
  }
...
}
```

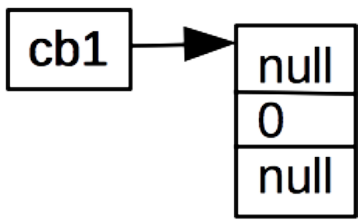
- La création peut avoir lieu dans un `main` ou dans une autre méthode.
- Le `main` peut être placé dans une autre classe.
- On fera en TD des classes ne contenant qu'une méthode `main` (classes-programme) et des classes ne servant qu'à représenter des concepts. Ici ce n'est pas fait pour raccourcir le code présenté sur la diapo.

Instance (ou objet) de `CompteBancaire`

```
CompteBancaire cb1 = new CompteBancaire();  // creation
```

À gauche représentation UML, à droite schéma qui peut vous servir à vous représenter ce qui se passe dans la mémoire du programme :





- les attributs contiennent des valeurs par défaut correspondant à leur type :
  - 0 pour les nombres
  - `null` pour les classes
  - Ici la classe est `String`. La valeur `null` ne doit pas être confondue avec la chaîne vide de caractères `""`
  - false pour les booléens

Accès aux attributs

On peut aussi :

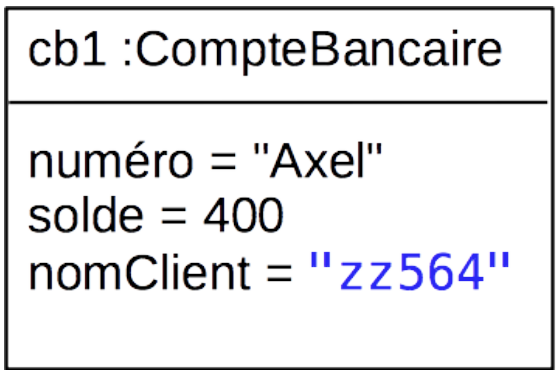
- accéder à leurs attributs grâce à l'opérateur . (point) à condition d'être dans la même classe

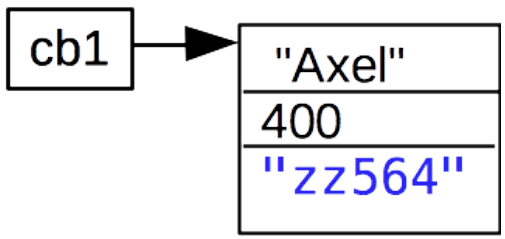
```
public class CompteBancaire {
  private String numero;
  private double solde;
  private String nomClient;

  public static void main(String[] a) {

    CompteBancaire cb1 = new CompteBancaire();  // creation

    cb1.nomClient = "Axel";           // acces par l'operateur .
    cb1.solde = 400;
    cb1.numero = "zz564";
  }
...
}
```

C'est possible ici car le `main` est dans la classe `CompteBancaire`.

Instance de CompteBancaire après modification des attributs

```
cb1.nomClient = "Axel";          // acces par l'operateur .
cb1.solde = 400;
cb1.numero = "zz564";
```

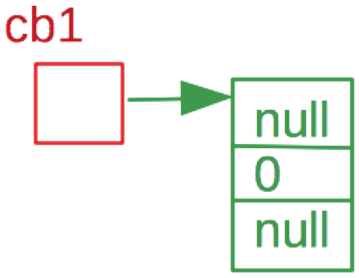




- les valeurs des attributs ont été mises à jour lors des affectations

Détails sur la déclaration et la création

```
CompteBancaire cb1 = new CompteBancaire();
```



Une autre écriture :

```
CompteBancaire cb1 ; // déclaration variable cb1
cb1 = new CompteBancaire(); // création objet
```

Pourquoi les attributs sont-ils privés ?

Règle métier : le découvert d'un compte ne peut dépasser 500 euros ... Mais on peut écrire :

```
cb1.solde = -10000;
```

Dans la classe elle-même, on peut y faire attention, mais dans d'autres programmes utilisateurs de la classe, ce sera plus difficile, donc on va contrôler ces affectations pour avoir des données cohérentes dans une méthode appelée accesseur (voir plus loin).

Partie comportementale : les méthodes

Les algorithmes ou opérations propres aux classes sont appelés des méthodes.

Constitution :

- signature (entête)
  - visibilité
  - type de la valeur retournée (ou `void` si rien n'est retourné)
  - nom
  - liste de paramètres
- corps
  - bloc d'instructions

```
public boolean debiter (double montant) { .... }
```

Partie comportementale : les méthodes

Une méthode (qui n'est pas static) s'applique à un objet ; on dit aussi que l'on envoie un message à l'objet.

- Pour appeler une méthode m sur un objet o avec le paramètre p, on écrit $o.m(p)$
- On n'écrit pas seulement $m(p)$, sinon on ne sait pas quel objet doit exécuter la méthode m

Pendant l'exécution de la méthode, l'objet auquel le message est envoyé (auquel la méthode est appliquée) est désigné par la pseudo-variable this.

Constructeurs

Les constructeurs sont des méthodes qui servent à initialiser les objets au moment de leur création.

Particularités :

- ils n'ont pas de type de retour
- ils portent le même nom que la classe
- les valeurs des paramètres sont utilisées pour initialiser les attributs

Constructeur sans paramètre (par défaut)

Il initialise un objet (une instance) avec des valeurs par défaut, cohérentes avec la signification de la classe.

```
public class CompteBancaire{
...

   public CompteBancaire() {
       this.nomClient = "client inconnu";
       this.numero = "numéro non affecté";
       // on ne met rien dans solde,
       // la valeur par défaut, qui est 0, nous convient
   }
...
}
```

Constructeur avec paramètre (constructeur 1)

Il initialise un objet (une instance) avec des valeurs passées en paramètre et si besoin complète avec des valeurs par défaut.

En voici un premier exemple : on initialise le nom du client avec une valeur passée en paramètre et le numéro avec une valeur littérale.

```
public CompteBancaire(String nomClient) {
  this.nomClient = nomClient;
  this.numero = "numéro non affecté";
  // on ne met rien dans solde, la valeur par défaut nous convient
}
```

Constructeur avec paramètre (constructeur 2)

Il initialise un objet (une instance) avec des valeurs passées en paramètre et si besoin complète avec des valeurs par défaut.

En voici un second exemple : on initialise le nom du client et le numéro avec des valeurs passées en paramètres.

```
public CompteBancaire(String nomClient, String numero) {
  this.nomClient = nomClient;
  this.numero = numero;
    // on ne met rien dans solde, la valeur par défaut nous convient
}
```

Accesseurs

Ce sont des méthodes d'accès qui seront nécessaires pour manipuler les attributs (privés) depuis des méthodes hors de la classe.
Ils vont souvent par paires :

- un accesseur en lecture pour connaître la valeur ; sa forme est get suivi du nom de l'attribut commençant par une majuscule
- un accesseur en écriture pour modifier la valeur. Il permettra notamment de contrôler la manière dont on la modifie ; sa forme est set suivi du nom de l'attribut commençant par une majuscule

```
public String getNumero(){return numero; }

public void setNumero(String numero) {this.numero = numero; }

public String getNomClient(){return nomClient; }

public void setNomClient(String nomClient)
                {this.nomClient = nomClient; }
```

Accesseurs

L'accesseur en écriture sera utilisé pour contrôler les valeurs affectées aux attributs, en cohérence avec les règles métier.

- Le solde ne doit jamais être négatif (supposons que l'on interdise les comptes débiteurs pour cet exemple)

```
public void setSolde(double nouveauSolde) {
  if (nouveauSolde <0)
   {System.out.println("erreur : un solde ne doit pas être négatif");}
  else
   {this.solde = nouveauSolde;}
}
```

Accesseurs

```
public void setSolde(double nouveauSolde) {
  if (nouveauSolde <0)
   {System.out.println("erreur : un solde ne doit pas être négatif");}
  else
   {this.solde = nouveauSolde;}
}
```

Instruction conditionnelle à 2 branches

- si le solde passé en paramètre est négatif, on affiche une erreur
- sinon la valeur du paramètre est bien écrite dans l'attribut solde de l'objet receveur du message, désigné par this

Accesseurs

Une autre version de la même méthode (il y en aura une seule dans le programme, on devra choisir entre les deux).

```
public void setSolde(double nouveauSolde) {
  if (nouveauSolde  >= 0)
   {this.solde = nouveauSolde;}
}
```

Instruction conditionnelle à 1 branche

- si le solde passé en paramètre est positif la valeur du paramètre est bien écrite dans l'attribut solde de l'objet receveur du message, désigné par this

sous-entendu : on ne fait rien dans les autres cas

Autres méthodes

La méthode toString() est une méthode ordinairement présente dans toutes les classes.

Elle retourne une représentation de l'objet sous forme d'une chaîne de caractères (String).

```
public String toString() {
   return "Client : "+this.nomClient+
     " Numéro : "+this.numero+" solde = "+this.solde;
}
```

Autres méthodes

La méthode verseIntérêts a pour paramètre un taux d'intérêt (taux entre 0 et 1)

Elle modifie le solde en ajoutant des intérêts correspondant au taux

Elle ne retourne rien (void)

```
public void verseIntérêts(double taux) {
  this.solde = this.solde * (1+taux);
}
```

Appel des constructeurs

```
public static void main(String[] args) {
   // Appel du constructeur 1
   CompteBancaire cb1 = new CompteBancaire("Marie");

   // Appel du constructeur 2
   CompteBancaire cb2 = new CompteBancaire("Sarah","ZZZ34");

....

}
```

Appel des méthodes : observer que la méthode est appliquée à un objet

```
public static void main(String[] args) {

....

   // Appel de la méthode toString
   System.out.println("cb1 "+cb1.toString());

   // on peut aussi écrire comme suit (appel implicite de toString)
   System.out.println("cb1 "+cb1);

....

// Appel de la méthode verseIntérêts
    cb1.verseIntérêts(0.01);
    cb2.verseIntérêts(0.05);
}
```

Appel de méthodes

```
public void setSolde(double nouveauSolde) {
  if (nouveauSolde  >= 0)
   {this.solde = nouveauSolde;}
}

public static void main(String[] args) {
   CompteBancaire cb1 = new CompteBancaire("Marie");
   CompteBancaire cb2 = new CompteBancaire("Sarah","ZZZ34");
   // Modification des soldes
   cb1.setSolde(400);
   cb2.setSolde(500);
   ....
}
```

À comprendre :

- pendant l'exécution de cb1.setSolde(400) ;
  this est cb1 ; l'attribut solde de cb1 est modifié
- pendant l'exécution de cb2.setSolde(500) ;
  this est cb2 ; l'attribut solde de cb2 est modifié

Synthèse

- Comprendre l'intérêt des classes qui regroupent structure et comportement d'un ensemble d'objets similaires
- Définition de la structure, composée d'attributs
- Définition du comportement, composé de méthodes
- Création d'objets
- Comprendre comment on appelle une méthode, par envoi de message à un objet (ou encore on dit application de la méthode à l'objet)
- Comprendre la pseudo-variable this
- Instructions conditionnelles