

Secteur Tertiaire Informatique  
Filière « Etude et développement »

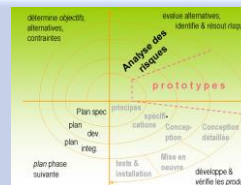
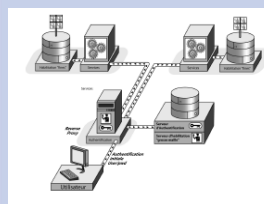
**Gestion de comptes bancaires**

**Programmation Orientée Objet**

Apprentissage

Mise en situation

Evaluation



# 1. INTRODUCTION

L'objectif de ce TP est de concevoir un programme en console basé sur une approche objet et permettant de gérer les comptes en banque de client.

Vous allez créer deux classes :

- « **Account** » : représente des comptes bancaires
- « **Customer** » : représente des clients d'une banque

Avec ces classes vous pourrez implémenter les fonctionnalités suivantes :

- Créer des comptes bancaires (avec vérification de leur IBAN)
- Créer des clients
- Associer des comptes bancaires aux clients

Le code attendu doit suivre les règles syntaxiques et les conventions de nommage du langage Java

Pour vous aider dans le développement une base de code est disponible à l'adresse suivante : <https://github.com/afpa-learning/poo-account>

## 2. IMPLEMENTATION DES CLASSES

### 2.1 IMPLEMENTATION DE LA CLASSE « ACCOUNT »

Dans un premier temps, il vous faudra créer une classe représentant un compte bancaire.

Un compte bancaire sera défini par les caractéristiques suivantes :

- **IBAN** : International Bank Account Number, code unique international d'un compte, de type chaîne de caractères ;
- « **balance** » (**montant**) : argent sur le compte (peut être négatif) , entier
- « **overdraftAuthorization** » : découvert autorisé, entier

Dans ce projet, nous considérerons que les montants des comptes sont en valeur entières.

Cette première version de la classe employée peut être graphiquement représentée en utilisant le langage UML comme présenté ci-dessous:

Account
<ul style="list-style-type: none"> <li>- iban: String</li> <li>- balance: int</li> <li>- overdraftAuthorization: int</li> </ul>
<ul style="list-style-type: none"> <li>+ constructeur</li> <li>+ getters</li> <li>+ setters</li> <li>+ toString():</li> <li>+ addMoney(amount: int): int</li> <li>+ removeMoney(amount: int): int</li> <li>+ transfer(otherAccount: Account, amount: int): void</li> </ul>

Quelques détails sur les méthodes à implémenter :

- « addMoney » : permet d'ajouter un montant sur le compte
- « removeMoney » : permet de retirer un montant du compte (si possible)
- « transfer » : prend en paramètre un **deuxième** objet de la classe « Account » et de transférer un certain montant
- 

### Attention

Si un retrait d'argent dépasse l'autorisation de découvert l'opération ne devra pas être possible !

Un message devra être affiché en console.

### A faire

Complétez la classe « Account » pour y ajouter tous les attributs ainsi que les « **getters** » (appelés accesseurs en français) et les « **setters** » (appelés mutateurs) et implémentez les méthodes.

### A faire

Ajoutez la méthode « **toString()** » qui aura pour objectif de transformer un objet en une représentation sous forme de chaîne de caractères.

Pour plus d'information concernant la méthode « **toString()** » :

<https://codegym.cc/fr/groups/posts/fr.986.mthode-java-tostring>

## A faire

**Instanciez** quelques objets de la classe « Account » avec des informations différentes et essayez toutes les fonctionnalités.

### 2.1.1 Vérification de l'IBAN

## A faire

Ajoutez, dans la classe « Account » une méthode de vérification de l'IBAN nommée « **checkIban** ».

Voilà un exemple de déclaration que vous pourrez utiliser pour cette méthode :

```
public boolean checkIban(String stringToCheck)
```

Type de retour : **boolean** → indique si l'objet de type « String » passé en paramètre est un IBAN ou pas

Paramètre d'entrée : **String stringToCheck** → la chaîne de caractère à vérifier

Testez cette méthode avec

Plus d'information sur l'algorithme de vérification d'IBAN disponible ici : [https://fr.wikipedia.org/wiki/International\\_Bank\\_Account\\_Number](https://fr.wikipedia.org/wiki/International_Bank_Account_Number)

Jusqu'à présent nous avons vu que pour **appeler** une **méthode** il **fallait forcément utiliser un objet**.

Dans certain cas cette approche n'est pas la plus pertinente.

Prenons l'exemple de la méthode « **checkIban** » :

```
// instantiation d'un compte avec l'IBAN "FR76 3000 1007 9412 3456 78901 85"
Account account1 = new Account("FR7630001007941234567890185", 5000);

// vérification de l'IBAN "FR76 4255 9000 0112 3456 78901 21"
boolean isCorrectIban = account1.checkIban("FR7642559000011234567890121");
```

Dans ce cas nous utilisons un objet de classe « **Account** » représentant le compte associé à l'IBAN « **FR76 3000 1007 9412 3456 78901 85** » pour vérifier un IBAN associé à un autre compte.

Ceci peut paraître étrange et peut nous mener à la question suivante :

**Est-il nécessaire de pouvoir accéder aux attributs de « **account1** » pour vérifier ce nouvel IBAN (via le « this ») ?**

La réponse à cette question est : **nous n'en avons pas besoin.**

### Important

Il devrait être possible d'utiliser la méthode « checkIban » **sans avoir à utiliser un objet spécifique.**

Cette méthode peut être considérée comme **utilitaire. Elle n'a besoin que de son paramètre d'entrée pour fonctionner.**

Nous pouvons maintenant nous poser une autre question : **est-il pertinent de laisser la méthode « checkIban » dans la classe « Account » ?**

Pour répondre à cette question il faut nous questionner sur la **responsabilité de la classe.**

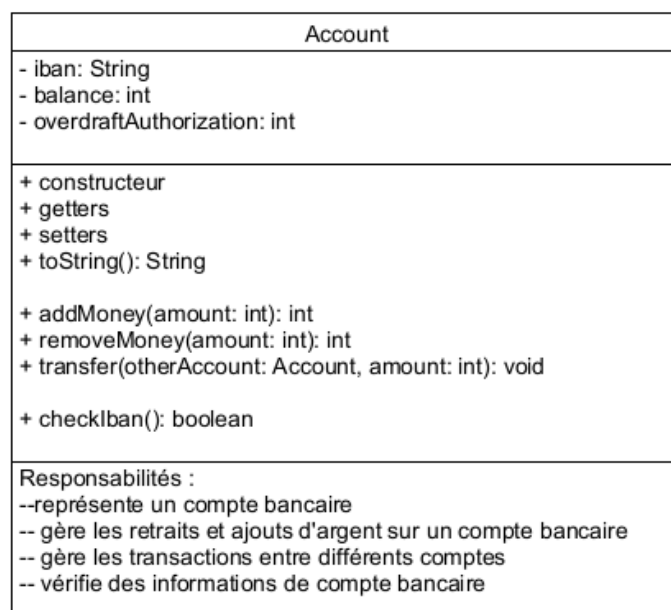
### Important

La **responsabilité** d'une classe **correspond à ce qu'elle est supposée faire.**  
Autrement dit, **ce pourquoi elle a été conçue.**

Dans notre cas la classe s'appelle « **Account** », nous allons lui donner les responsabilités suivantes :

- représente un compte bancaire ;
- gère les retraits et ajouts d'argent sur un compte bancaire ;
- gère les transactions entre différents comptes ;
- vérifie des informations de compte bancaire.

Nous pouvons ajouter les responsabilités d'une classe sur le diagramme UML comme présenté par la figure suivante :



Avec cette modélisation il est donc pertinent de mettre la méthode « **checkIban** » dans la classe « **Account** », cela fait partie de ses responsabilités.

### Important

Il n'existe pas de **solution absolue et totalement évidente** en ce qui concerne l'organisation du code, tout dépend de votre vision des choses et de votre façon de modéliser un problème.

Cet aspect fait partie de votre futur métier **d'analyste** programmeur.

Nouvelle question à se poser : mais si nous choisissons de laisser la méthode « **checkIban** » dans la classe « **Account** » mais que celle-ci **ne concerne aucune instance de classe spécifique**, comment faisons-nous ?

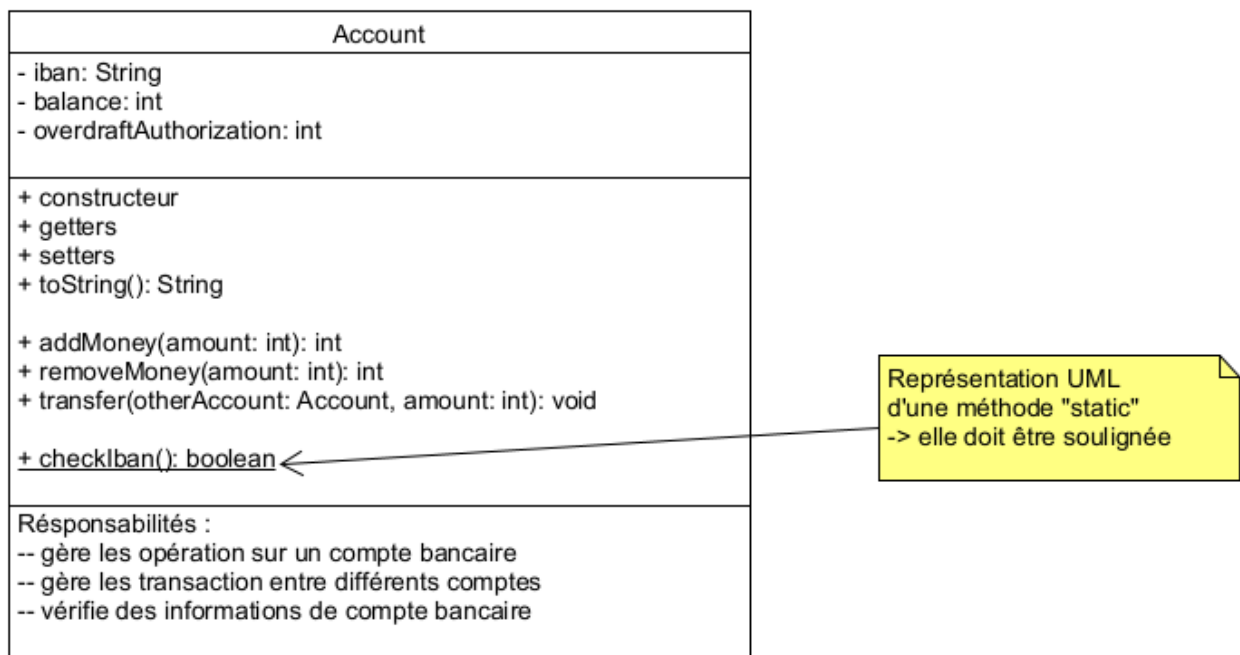
Nous allons déclarer la méthode « **checkIban** » comme étant « **static** »

Dans une classe, un attribut ou une méthode « static » sont des éléments que l'on peut **appeler sans instance de classe (sans objet)**.

En Java, il est possible de la déclarer comme suit (l'œil averti repérera le mot clef « static ») :

```
public static boolean checkIban(String stringToCheck)
```

Sa représentation UML est alors :



**L'appel d'une fonction « static » peut être effectué en utilisant uniquement le nom de la classe, comme suit :**

```
// vérification de l'IBAN "FR76 4255 9000 0112 3456 78901 21"
boolean isCorrectIban = Account.checkIban("FR7642559000011234567890121");
```

Nous observons qu'il n'est plus nécessaire d'utiliser un objet spécifique, **seulement le nom de classe est nécessaire**.

## Récapitulatif

Un attribut ou une méthode « **non-static** » appartient à une instance de classe (un objet).

D'un autre côté, un attribut ou une méthode « **static** » appartient à la classe en elle-même.

## 2.2 IMPLEMENTATION DE LA CLASSE « CUSTOMER »

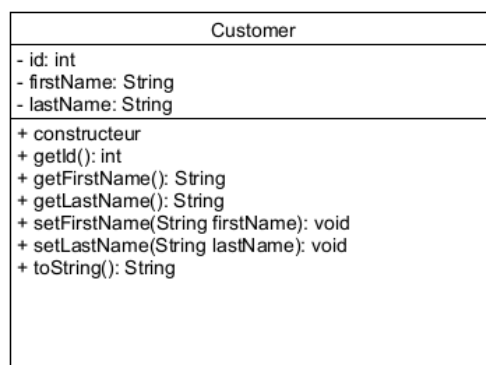
### 2.2.1 Première version

Vous allez créer une seconde classe qui sera en relation avec « Account », la classe « Customer ».

Un « Customer » est défini par les caractéristiques suivantes :

- Id : identifiant unique, entier
- « **firstName** » : prénom, string
- « **lastName** » : nom, string

Première représentation UML :

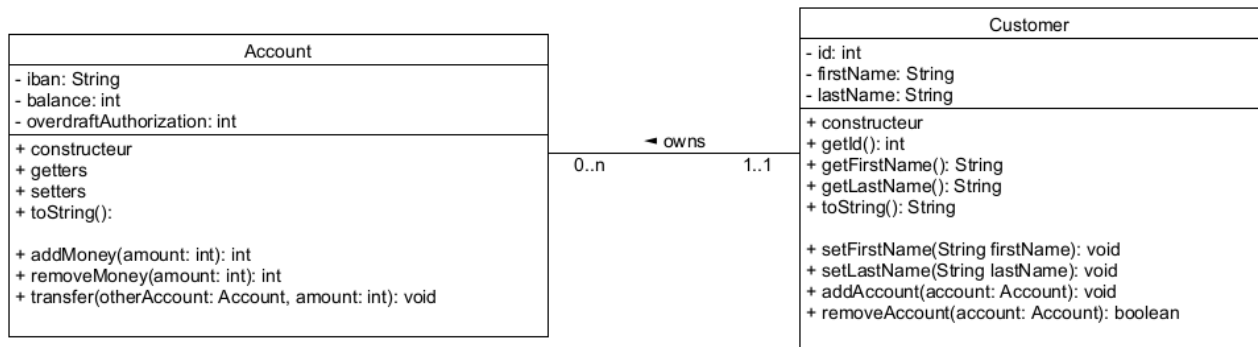


## A faire

Implémentez une première version de cette classe, comme présenté par le diagramme UML.

### 2.2.2 Relation entre « Customer » et « Account »

Vous allez maintenant implémentez la relation entre un « Customer » et un compte bancaire, comme présenté par la figure suivante :



Règle métier indiquée sur ce diagramme : un client peut avoir un ou plusieurs comptes bancaires mais peut aussi ne pas en avoir.

En UML, cette règle est indiquée par le lien « **owns** » qu'il y a entre « la classe « **Customer** » et « **Account** » et les **cardinalités** de chacun des liens :

- **0..n** : indique que le client peut avoir de **0 à un nombre indéterminé** de comptes bancaire
- **1..1** : indique qu'un compte bancaire ne **peut avoir qu'un seul client**

Pour implémenter une telle contrainte vous allez pouvoir ajouter un **attribut** de type « **ArrayList<Account>** » dans la classe « **Customer** », par exemple avec le code suivant :

```
private ArrayList<Account> accounts = new ArrayList<Account>();
```

Vous pouvez maintenant ajouter les méthodes d'ajout et de suppression des comptes :

- « **addAccount(Account account)** »
- « **removeAccount(Account account)** »

Ces méthodes devront faire appel à des méthodes de la classe « **ArrayList** ».

Pour plus d'informations sur la classe « **ArrayList** » veuillez-vous référer au site suivant : [https://www.w3schools.com/java/java\\_arraylist.asp](https://www.w3schools.com/java/java_arraylist.asp)

## A faire

Une fois la « **ArrayList** » ajoutée et les méthodes implémentées testez votre code à partir de la fonction « **main** » en instanciant plusieurs objets des classes « **Account** » et « **Customer** » et en ajoutant/supprimant des comptes à un client en utilisant les méthodes « **addAccount** » et « **removeAccount** ».



## **CREDITS**

### **ŒUVRE COLLECTIVE DE L'AFPA**

**Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services**

### **Equipe de conception (IF, formateur, mediatiseur)**

Michel Coulard – Formateur Evry

Chantal Perrachon – IF Neuilly sur Marne

**Date de mise à jour : 23/06/2024**

## **Reproduction interdite**

Article L 122-4 du code de la propriété intellectuelle.

« Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction par un art ou un procédé quelconque. »