1. Gestion des comptes bancaires : la classe Account

*Chaque **compte** bancaire a un numéro unique (`number`) qui lui est attribué dès l'ouverture et ne peut être modifié par la suite. Le compte appartient à une personne, titulaire du compte (`name`), qui dispose d'un solde (`balance`) en banque pour ce compte. Pour garantir l'intégrité des données des comptes nous mettons en œuvre le concept **d'encapsulation** en **masquant** les données.*

a. Créer un nouveau **projet** (`TP2Bank`), **sous eclipse**, et un nouveau package (`bank`) dans ce projet. Ce package contiendra toutes vos classes métier. Ajouter y une nouvelle classe `Account`.

```
package bank;
import … ;
public class Account {
   private String number;
   private String name;
   private double balance;

   // A COMPLETER
}
```

Account
- number : String - name : String - balance: double
+ deposit (double) **+ withdraw (double)** **+ transfer (Account, double)** **+ getters/setters** **+ equals()** **+ toString()**

b. Définir les **constructeurs** suivants pour la classe `Account` :

`public Account(String name, double balance)`	• le nom est saisi au clavier s'il est null
`public Account()`	• appèle le 1è constructeur
`public Account(Account original)`	• le constructeur par copie

Indication. *Le numéro du compte doit être unique et généré automatiquement à la création. Une façon de faire consiste à utiliser la méthode `randomUUID()` de la classe `UUID` (Universally Unique Identifier), comme suit :*

```
//Generates 16 bytes that contain hexadecimal values
String [] string = java.util.UUID.randomUUID().toString().split("-");
this.number = string[0].substring(4);
```

c. Définir les **méthodes d'instances** de la classe **`Account`** dans leur version minimale (sans gérer les exceptions) :

`public void deposit (final double amount)`	**Déposer** un montant donné dans le compte
`public void withdraw(final double amount)`	**Retirer** un montant donné du compte
`public void transfer(double amount,Account other)`	Faire un **virement** vers un autre compte

d. Redéfinir (`@Override`) les **méthodes usuelles** suivantes pour la classe **`Account`** :

`public String toString()`	Renvoie la représentation du compte sous forme d'une chaîne
`public boolean equals(Object)`	Deux comptes égaux si même `propriétaire` et même `numéro`
`getters/setters`	`setBalance(balance)` lève une exception si `solde < 0`

2. Tester la classe Account

Implémentez le **scénario** suivant en complétant `Main` (une classe séparée) :

a. **C**réez deux comptes bancaire (objets), effectuez ensuite les opérations suivantes en adaptant la méthode **`toString()`** pour l'affichage comme ci-contre !

- Déposer `3000.0` dhs `account1`
- Retirer `1000.0` dhs de `De account2`
- Virer `2000.0` dhs de `account1` vers `account2`

Console

```
IBAN= b503 Name= Customer1   Balance= 2000,00
IBAN= ce14 Name= Customer2   Balance= 2000,00
```

```
public class Main {
   public static void main(String[] args) {
       var account1 = new Account("Customer1", 1000.0);
       var account2 = new Account("Customer2", 1000.0);
       // A COMPLETER …
   }
}
```

b. Le retrait d'argent ou le virement n'est pas toujours possible. Il faut gérer également les **exceptions**, on commencera par définir la classe d'exception : `InsufficientFundException.`

```
public class InsufficientFundException extends Exception {
   public InsufficientFundException(String message) {
       super(message);
   }
}
```

c. Créez maintenant le compte `account1` avec comme solde initial `1000 Dhs` et essayer de faire un virement de `2000 Dhs` vers le compte `account2`.

Console

```
InsufficientFundException : credit limit exceeded
(Customer1 2000,00)
IBAN=06d2    Name=Customer1      Balance=1000,00
```