<?php //POO ?>

Prérequis

La lecture de ce document nécessite une connaissance des bases de la programmation :

- Notions de variable et de type.
- Structures conditionnelles et itératives (boucles).
- Programmation modulaire (sous-programmes) et passage de paramètres.
- Tableaux.

Introduction

La programmation orientée objet, souvent abrégée **POO**, permet de concevoir une application sous la forme d'un ensemble de briques logicielles appelées des **objets**. Chaque objet joue un rôle précis et peut communiquer avec les autres objets. Les interactions entre les différents objets vont permettre à l'application de réaliser les fonctionnalités attendues.

La POO facilite la conception de programmes par réutilisation de composants existants, avec tous les avantages évoqués plus haut. Elle constitue le standard actuel (on parle de paradigme) en matière de développement de logiciels.

Première approche de la POO

La notion d'objet

Quand on utilise la POO, on cherche à représenter le domaine étudié sous la forme d'objets. C'est la phase de **modélisation orientée objet**.

Un **objet** est une entité qui représente (modélise) un élément du domaine étudié : une voiture, un compte bancaire, un nombre complexe, une facture, etc.

Objet = état + actions

Cette équation signifie qu'un objet rassemble à la fois :

- des **informations** (ou données) qui le caractérisent.
- des **actions** (ou traitements) qu'on peut exercer sur lui.

Imaginons, par exemple, qu'on souhaite modéliser des comptes bancaires pour un logiciel de gestion. On commence par réfléchir à ce qui caractérise un compte bancaire, puis on classe ces éléments en deux catégories :

- les informations liées à un compte bancaire.
- les actions réalisables sur un compte bancaire.

En première approche, on peut considérer qu'un compte bancaire est caractérisé par un **titulaire**, un **solde** (le montant disponible sur le compte) et utilise une certaine **devise** (dinars, euros, dollars, etc). Les actions réalisables sur un compte sont le dépôt d'argent (**crédit**) et le retrait (**débit**).

La notion de classe

Nous venons de voir que l'on pouvait représenter un compte bancaire sous la forme d'un objet. Imaginons que nous voulions gérer les différents comptes d'une banque. Chaque compte aura son propre titulaire, son solde particulier et sa devise. Mais tous les comptes auront un titulaire, un solde et une devise, et permettront d'effectuer les mêmes opérations de débit/crédit. Chaque objet "compte bancaire" sera construit sur le même modèle : ce modèle est appelée une **classe**.

DEFINITION : une **classe** est un **modèle d'objet**. C'est un nouveau type créé par le programmeur et qui sert de modèle pour tous les objets de cette classe. Une classe spécifie les informations et les actions qu'auront en commun tous les objets qui en sont issus.

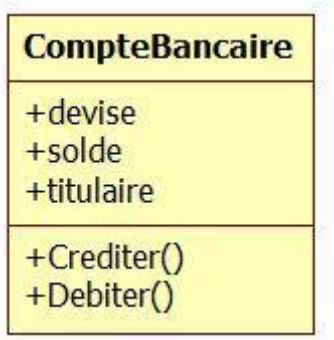
Le compte en banque appartenant à Jean, dont le solde est de 450 euros, est un compte bancaire particulier. Il s'agit d'un objet de la classe "compte bancaire". En utilisant le vocabulaire de la POO, on dit que l'objet "compte bancaire de Jean" est une **instance** de la classe "compte bancaire".

ATTENTION : ne pas confondre **objet** et **classe**. Une classe est un type abstrait (exemple : un compte bancaire en général), un objet est un exemplaire concret d'une classe (exemple : le compte bancaire de Jean).

Un objet est une variable particulière dont le type est une classe.

Représentation graphique

Afin de faciliter la communication entre les programmeurs n'utilisant pas le même langage de programmation objet, il existe un standard de représentation graphique d'une classe. Ce standard fait partie de la norme **UML (Unified Modeling Language)**. On parle de **diagramme de classe**.



Programmer avec des objets

Ecriture d'une classe

Une fois la modélisation d'une classe terminée, il est possible de la traduire dans n'importe quel langage de programmation orienté objet.

Voici la traduction en PHP de la classe **CompteBancaire**

```php
<?php
    class CompteBancaire{

        public $titulaire;
        public $solde = 100;
        public $devise;

        public function debiter($montant){
            $this->solde-=$montant;
        }

        public function crediter($montant){
            $this->solde+=$montant;
        }

         public function afficherSolde(){
            echo "<br>Votre solde est: ".$this->solde." ".$this->devise;
        }

    }
?>
```

La définition d'une classe commence par le mot-clé class. On retrouve ensuite la définition des champs (attributs) et des méthodes de la classe. On remarque que les méthodes utilisent (et modifient) les valeurs des attributs.

Une méthode est une sorte de mini-programme. On peut y déclarer des variables locales et y utiliser tous les éléments de programmation déjà connus (alternatives, boucles, etc).

On remarque un nouveau mot-clé : public, sur lequel nous reviendrons ultérieurement.

Utilisation d'une classe

En utilisant le modèle fourni par la classe, il est possible de créer autant d'objets que nécessaire. Différents objets d'une même classe disposent des mêmes attributs et des mêmes méthodes, mais les valeurs des attributs sont différentes pour chaque objet. Par exemple, tous les comptes bancaires auront un solde, mais sauf exception, ce solde sera différent pour chaque compte.

Le programme PHP ci-dessous utilise la classe **CompteBancaire** définie plus haut pour créer le compte de Ahmed et y effectuer deux opérations et affichage du solde du compte à chaque fois.

```php
<?php
    //appel du fichier contenant la classe
    include 'compteBancaire.php';

    //instanciation d'un nouvel objet
    $compteAhmed = new CompteBancaire();

    //affectation des valeurs aux attributs
    $compteAhmed->titulaire = "Ahmed Ben Salah";
    $compteAhmed->devise = "TND";

    //appels des méthodes
    $compteAhmed->afficherSolde();

    $compteAhmed->crediter(150);
    $compteAhmed->afficherSolde();

    $compteAhmed->debiter(50);
    $compteAhmed->afficherSolde();
?>
```

A la fin du programme l'attribut solde de l'objet **compteAhmed** contient la valeur 200 TND.

```
C:\wamp64\www\POO\bank-example\index.php:11:
object(compteBancaire)[1]
  public 'titulaire' => string 'Ahmed Ben Salah' (length=15)
  public 'solde' => int 100
  public 'devise' => string 'TND' (length=3)


Votre solde est: 100 TND
Votre solde est: 250 TND
Votre solde est: 200 TND
```

Instanciation

L'instanciation est l'opération qui consiste à créer un nouvel objet à partir d'une classe. L'instanciation se fait en utilisant le mot clé **new.**

```
5       //instanciation d'un nouvel objet
6       $compteAhmed = new CompteBancaire();
7
```

Cette instruction nous permet de créer un objet (variable) nommé **$compteAhmed** de type **CompteBancaire**.

Application

Enoncé

M. Ahmed Ben Salah souhaite effectuer un transfert d'argent de son compte bancaire vers celui de Mme. Imen Bejaoui titulaire, elle aussi, d'un compte dans la banque.

Ecrire le code nécessaire permettant de faire ce transfert.

Corrigé

Le fichier contenant le corps de la classe :

compteBancaire.php

```php
<?php
    class CompteBancaire{

        public $titulaire;
        public $solde = 100;
        public $devise;

        public function debiter($montant){
            $this->solde-=$montant;
        }

        public function crediter($montant){
            $this->solde+=$montant;
        }

         public function afficherSolde(){
            echo "<br>Le solde de ".$this->titulaire." est de: ".$this->solde." ".$this->devise;
        }

        public function transferer($beneficiaire, $mtTransfert){
            $beneficiaire->crediter($mtTransfert);
            $this->debiter($mtTransfert);
        }

    }
?>
```

Le fichier contenant les objets et leurs manipulations :

index.php

```php
<?php
    //appel du fichier contenant la classe
    include 'compteBancaire.php';

    //instanciation d'un nouvel objet
    $compteAhmed = new CompteBancaire();

    //affectation des valeurs aux attributs
    $compteAhmed->titulaire = "Ahmed Ben Salah";
    $compteAhmed->devise = "TND";
    var_dump($compteAhmed);

    //appels des méthodes
    $compteAhmed->crediter(500);
    $compteAhmed->afficherSolde();

    //création d'un nouveau compte
    $compteImen = new CompteBancaire();

    //affectation des valeurs aux attributs
    $compteImen->titulaire = "Imen Bejaoui";
    $compteImen->devise = "TND";
    var_dump($compteImen);


    //effectuer l'opération de transfert
    $compteAhmed->transferer($compteImen, 200);

    //affichage des nouveaux soldes
    $compteAhmed->afficherSolde();
    $compteImen->afficherSolde();

?>
```

Le résultat de l'exécution :

```
C:\wamp64\www\POO\bank-example\index.php:11:
object(CompteBancaire)[1]
  public 'titulaire' => string 'Ahmed Ben Salah' (length=15)
  public 'solde' => int 100
  public 'devise' => string 'TND' (length=3)


Le solde de Ahmed Ben Salah est de: 600 TND

C:\wamp64\www\POO\bank-example\index.php:23:
object(CompteBancaire)[2]
  public 'titulaire' => string 'Imen Bejaoui' (length=12)
  public 'solde' => int 100
  public 'devise' => string 'TND' (length=3)


Le solde de Ahmed Ben Salah est de: 400 TND
Le solde de Imen Bejaoui est de: 300 TND
```

Principaux concepts objets

Constructeur

Reprenons l'exemple de la classe `CompteBancaire`.

Tout compte a nécessairement un titulaire, un solde initial et une devise lors de sa création. On aimerait pouvoir instancier un objet de la classe `CompteBancaire` en définissant directement les valeurs de ses attributs. Pour cela, nous allons ajouter à notre classe une méthode particulière : **le constructeur**.

```php
class CompteBancaire{

    private $titulaire;
    private $solde;
    private $devise;

    //constructeur
    public function __construct($nomTitulaire, $soldeInitial, $laDevise){
        $this->titulaire = $nomTitulaire;
        $this->solde = $soldeInitial;
        $this->devise = $laDevise;
    }

```

Rôle

DEFINITION : le constructeur est une méthode spécifique dont le rôle est de construire un objet, le plus souvent en initialisant ses attributs.

ATTENTION : le nom du constructeur doit être identique à celui-ci : `__construct`, avec les des underscores au début.

L'utilisation d'un constructeur se fait au moment de l'instanciation de l'objet (opérateur `new`), en passant en paramètres les futures valeurs des attributs de l'objet créé.

```php
//Création du compte de Ahmed
$compteAhmed = new CompteBancaire("Ahmed Ben Salah",500,"TND");
//Création du compte de Imen
$compteImen = new CompteBancaire("Imen Bejaoui",150,"TND");
```

Constructeur par défaut

Lorsqu'une classe ne définit aucun constructeur, un constructeur par défaut sans aucun paramètre est implicitement créé. Il n'a aucun comportement mais son existence permet d'instancier des objets de cette classe.

En revanche, toute définition explicite d'un constructeur dans une classe "désactive" le constructeur par défaut. Dans notre exemple actuel, on ne peut plus instancier un compte bancaire sans lui fournir les trois paramètres que son constructeur attend.

Encapsulation

L'écriture de classes offre d'autres avantages que le simple regroupement de données et de traitements. Parmi ceux-ci figure la possibilité de restreindre l'accès à certains éléments de la classe. C'est ce que l'on appelle **l'encapsulation**.

Exemple d'utilisation

On souhaite qu'un compte bancaire créé ne puisse pas changer de titulaire ni de devise. Cela est possible en définissant les attributs `$titulaire` et `$devise` comme étant **privés**.

```php
class CompteBancaire{

    private $titulaire;
    public $solde;
    private $devise;

    //constructeur
    public function __construct($nomTitulaire, $soldeInitial, $laDevise){
        $this->titulaire = $nomTitulaire;
        $this->solde = $soldeInitial;
        $this->devise = $laDevise;
    }
```

A présent, la seule manière de définir des valeurs pour `$titulaire` et `$devise` est d'utiliser le constructeur. Toute tentative d'accès externe aux propriétés privées génèrera une erreur.

Définition

Les mots-clés `public` et `private` permettent de modifier **le niveau d'encapsulation** (on parle aussi de **visibilité** ou d'**accessibilité**) des éléments de la classe (attributs et méthodes) :

- un élément **public** est librement utilisable depuis le reste du programme.
- un élément **privé** est uniquement utilisable depuis les méthodes de la classe elle-même.

REMARQUE : dans de nombreux langages dont le PHP, il existe un niveau d'encapsulation intermédiaire (`protected`) qui ne fera pas partie de ce cours.

DEFINITION : l'**encapsulation** est l'un des principes fondamentaux de la POO. Il consiste à restreindre l'accès à certains éléments d'une classe (le plus souvent ses attributs). L'objectif de l'encapsulation est de ne laisser accessible que le strict nécessaire pour que la classe soit utilisable.

CONSEIL : sauf cas particulier, on donne le niveau de visibilité `private` à tous les attributs d'une classe afin d'assurer leur encapsulation par défaut.

```php
class CompteBancaire{

    private $titulaire;
    private $solde;
    private $devise;

    //constructeur
    public function __construct($nomTitulaire, $soldeInitial, $laDevise){
        $this->titulaire = $nomTitulaire;
        $this->solde = $soldeInitial;
        $this->devise = $laDevise;
    }
```

Avantages

L'encapsulation offre de nombreux avantages :

- diminution des risques de mauvaise manipulation d'une classe.
- création de classes "boîtes noires" par masquage des détails internes.
- possibilité de modifier les détails internes d'une classe (la manière dont elle fonctionne) sans changer son comportement extérieur (ce qu'elle permet de faire).

Accesseurs

L'encapsulation des attributs a permis d'interdire toute modification (accidentelle ou volontaire) des données d'un compte bancaire. Cependant, il est maintenant impossible de consulter le solde, le titulaire ou la devise d'un compte créé, ce qui est gênant. On aimerait pouvoir accéder aux données de la classe, tout en maintenant un certain niveau de contrôle. Cela est possible en ajoutant des **accesseurs** à la classe.

DEFINITION : un accesseur est une méthode le plus souvent publique qui permet d'accéder à un attribut privé.

- un accesseur en lecture (**getter**) permet de lire la valeur d'un attribut.
- un accesseur en écriture (mutateur ou **setter**) permet de modifier la valeur d'un attribut.

Par exemple, voici la liste des accesseurs de notre classe `CompteBancaire` :

```php
public function getTitulaire(){
    return "<br>Le titulaire de ce compte est: ".$this->titulaire;
}

public function getSolde(){
    return "<br>Le solde de ".$this->titulaire." est de: ".$this->solde." ".$this->devise;
}

public function getDevise(){
    return "<br>La devise utilisée sur le compte de ".$this->titulaire." est la: ".$this->devise;
}

public function setTitulaire($newNom){
    $this->titulaire = $newNom;
}

public function setDevise($newDevise){
    $this->devise = $newDevise;
}
```

REMARQUE : on a choisi de ne définir un setter que pour l'attribut `$solde`, ce produit ne peut être modifiable (après la création du compte avec un solde initial) qu'avec les transactions possibles sur un compte. On parle alors d'un attribut en **lecture seule**.