Cours : Programmation orientée objet en C#
Cote du cours : 26978
Session : Hiver 2022

Chapitre 11

Les classes et méthodes abstraites

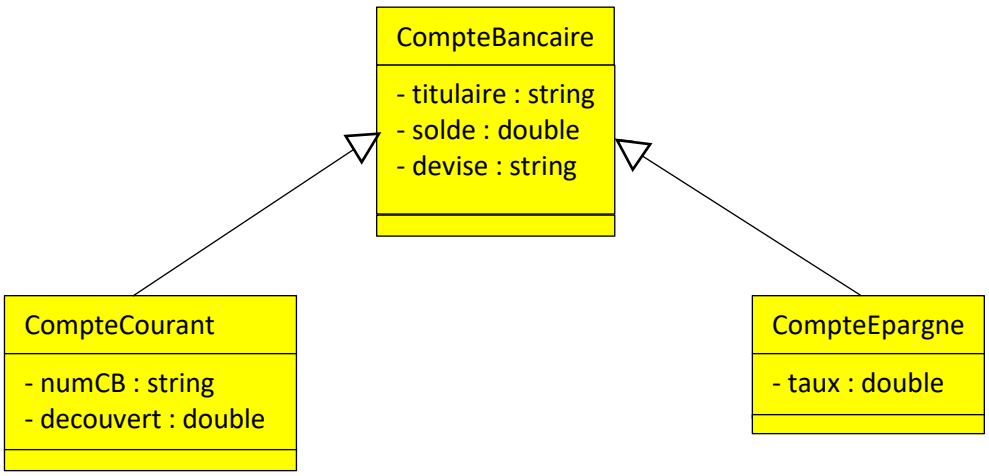
Table des matières

Introduction

La notion d'Héritage apporte avec elle plusieurs défis lors de la définition des comportements. Un de ces défis est les classes et méthodes abstraites. Pour bien comprendre ce principe, nous allons considérer l'exemple des comptes bancaires.

- Un compte bancaire est caractérisé par un *titulaire* (string), le *solde* (double) et la *devise* (string).
- Un compte bancaire est soit un compte courant, soit un compte épargne.
- Un *compte courant* se caractérise par le *numéro* de la carte bancaire qui lui est associée, ainsi que par un *découvert* maximal autorisé.
- Un *compte épargne* est caractérisé par un *taux* d'intérêt.

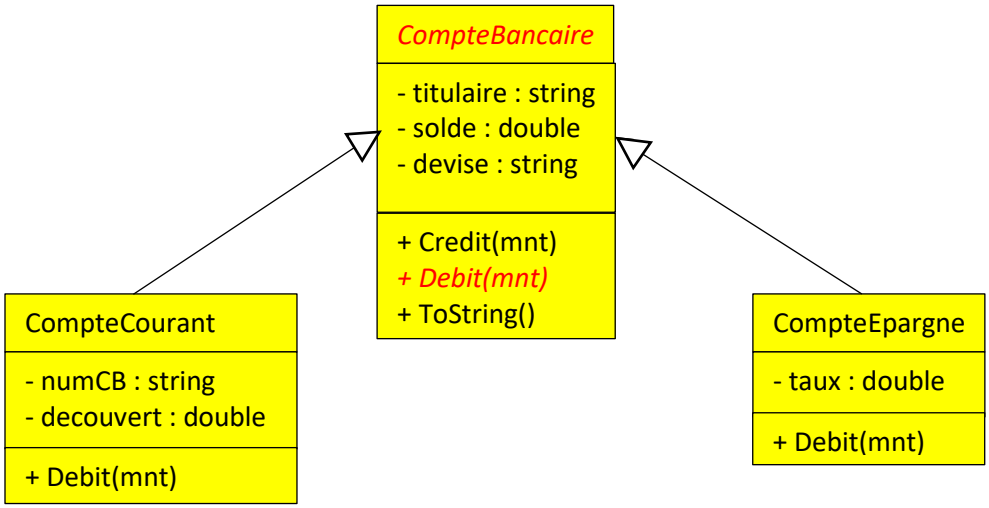
Une modélisation de la solution se présente dans le diagramme de classe suivant:



Nous voulons maintenant modéliser les comportements suivants :

- Les comptes bancaires disposent d'une opération de dépôt (crédit) et de retrait (débit). Ils fournissent aussi une description du compte.
- Pour les comptes courants, l'opération de retrait qui ferait passer le nouveau solde en dessous du découvert maximal est interdite et non effectuée.
- Pour les comptes épargnes, on ne peut retirer en une seule fois plus de la moitié du solde du compte.

Dans cette description, on remarque que les opérations de dépôt d'argent et de retour de la description du compte fonctionnent de la même manière pour tous les types de comptes. Tandis que pour l'opération de retrait, je ne peux implémenter cette méthode que lorsque je connais le type de compte en question. Pour traiter cette problématique, il faudrait pouvoir déclarer une opération de débit dans la superclasse CompteBancaire et laisser les classes dérivées définir comment cette opération est effectuée. Pour réaliser cette contrainte, on utilise une **méthode abstraite** (représenter en italique dans le digramme ci-dessous).



I. Les classes abstraites

1. Définition

Les classes abstraites sont utilisées pour centraliser des éléments de code communs à plusieurs types ou pour fournir des fonctionnalités de base devant être complétées dans un type dérivé (définition d'un concept).

Les classes abstraites sont avant tout des classes. Ils peuvent donc contenir tous les éléments de code d'une classe dite concrète (non abstraite) : méthodes, propriétés, variables membres...

Une méthode abstraite est une méthode dont l'implémentation n'est pas fournie.

Une **classe abstraite** définit un concept abstrait, incomplet ou théorique. Elle rassemble des éléments **communs** à plusieurs classes dérivées. **Elle n'est pas destinée à être instanciée**.

2. Propriétés

- Les classes abstraites sont des types définis comme non instanciables.
- Les classes abstraites ne sont utilisables qu'au travers des classes dérivées.
- Les classes abstraites **peuvent** contenir des méthodes abstraites. Il est alors nécessaire que chaque type dérivé fournisse sa propre implémentation de la méthode.
- Une classe comportant au moins une méthode abstraite est nécessairement une **classe abstraite**.

3. Syntaxe

Pour déclarer une classe abstraite on utilise le mot clé abstract.

```
<modificateur accès> abstract class NomClasse { }
```

Pour déclarer une méthode abstraite dans une classe abstraite on utilise aussi le mot clé abstract.

<modificateur accès> **abstract typeRetour** Méthode(arg1, arg2,…) **;**

Exemple complet :

```
public abstract class ClasseAbstraite
{
    public abstract void Methode();
}
public class ClasseDerivee : ClasseAbstraite
{
    public override void Methode() { }
}
```

En C#, la redéfinition d'une méthode abstraite doit être précédée du mot-clé **override**.

II. Exemple CompteBancaire

Ci-dessous le corrigé de l'exemple du compte bancaire et ses sous classes :

```
public abstract class CompteBancaire
{
    private string titulaire;
    public string Titulaire {
        get { return this.titulaire; }
        set { this.titulaire = value; }
    }

    private double solde;
    public double Solde
    {
        get { return this.solde; }
        set { this.solde = value; }
    }

    private string devise;
    public string Devise
    {
        get { return this.devise; }
        set { this.devise = value; }
    }

    public CompteBancaire(string titulaire, double solde, string
devise)
```

```
    {
            this.Titulaire = titulaire;
            this.Solde = solde;
            this.Devise = devise;
    }
    public override string ToString()
    {
        return "Titulaire : " + this.Titulaire + "\n" +
                "Solde : " + this.Solde + " "+this.Devise;
    }

    public void Credit(double mnt)
    {
        Solde += mnt;
    }

    public abstract void Debit(double mnt);
}

public class CompteCourant : CompteBancaire
{
    private string numCB;
    public string NumCB
    {
        get { return this.numCB; }
        set { this.numCB = value; }
    }

    private double decouvert;
    public double Decouvert
    {
        get { return this.decouvert; }
        set { this.decouvert = value; }
    }
    public CompteCourant(string titulaire, double solde, string
devise, string numCB, double decouvert): base(titulaire, solde,
devise)
    {
        this.NumCB = numCB;
        this.Decouvert = decouvert;
    }
    public override void Debit(double mnt)
    {
        if (mnt <= (this.Solde + this.Decouvert))
            Solde -= mnt;
    }
```

```
}
public class CompteEpargne : CompteBancaire {
    private double taux;
    public double Taux
    {
        get { return this.taux; }
        set { this.taux = value; }
    }

    public CompteEpargne(string titulaire, double solde, string
devise, double taux) : base(titulaire, solde, devise)
    {
        this.Taux = taux;
    }
    public override void Debit(double mnt)
    {
        if (mnt < Solde/2)
            Solde -= mnt;
    }
}
```

Un compte bancaire est soit un compte courant, soit un compte épargne. Un compte bancaire en général n'a pas **d'existence concrète**. La classe CompteBancaire est une **abstraction** destinée à factoriser ce qui est commun à tous les comptes, mais pas à être instanciée.

```
static void Main(string[] args)
{
    CompteCourant c1 = new CompteCourant("Mohamed", 1500, "$CAD",
"123456789", 1000);
    Console.WriteLine(c1);
```

```
Titulaire : Mohamed
Solde : 1500 $CAD
```

```
    c1.Debit(200);
    Console.WriteLine(c1);
```

```
Titulaire : Mohamed
Solde : 1300 $CAD
```

```
    c1.Debit(3000);
    Console.WriteLine(c1);
```

```
Titulaire : Mohamed
Solde : 1300 $CAD
```

```
    CompteEpargne c2 = new CompteEpargne("Jaque", 25000, "$CAD", 13);

    Console.WriteLine(c2);
```

```
Titulaire : Jaque
Solde : 25000 $CAD
```

```
    c2.Debit(100);
    Console.WriteLine(c2);
```

```
Titulaire : Jaque
Solde : 24900 $CAD
```

```
    c2.Debit(13000);
    Console.WriteLine(c2);

}
```

```
Titulaire : Jaque
Solde : 24900 $CAD
```

III. Le transtypage

1. Cast Implicite et explicite

Le transtypage est un des points fondamentaux de la programmation orientée objet avec C#, car il est une des clés pour la bonne mise en œuvre du polymorphisme. C'est en effet lui qui permet de visualiser un objet sous plusieurs formes.

En fonction du contexte d'utilisation, il peut être implicite ou explicite. Les conversions implicites sont effectuées automatiquement lorsqu'elles sont nécessaires. Elles ne peuvent être effectuées que lorsque deux conditions sont réunies :

- Les types source et destination sont compatibles : les deux types sont liés par une relation d'héritage.
- La conversion d'un type vers l'autre n'entraîne aucune perte de données.

Des transtypages implicites peuvent être effectués d'un type dérivé vers un type de base. Cette opération est possible car l'objet du type dérivé est forcément un objet du type de base.

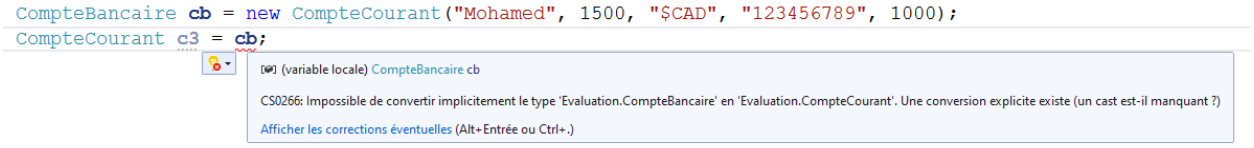
Exemple :

Un compte courant est forcément un compte bancaire. Je peux sauvegarder un objet de type compte courant dans un objet compte bancaire sans aucune intervention de ma part.

```
CompteCourant c1 = new CompteCourant("Mohamed", 1500, "$CAD",
"123456789", 1000);

CompteBancaire cb = c1;
```

En revanche, le passage d'un objet Compte bancaire vers un objet compte courant n'est pas aussi évidente. Cette opération n'est valide que si l'objet compte bancaire contient déjà un objet Compte courant.



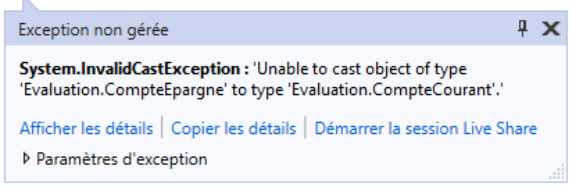
Pour corriger cette erreur de compilation, on fait une opération de cast.

```
CompteBancaire cb = new CompteCourant("Mohamed", 1500, "$CAD",
"123456789", 1000);
CompteCourant c3 = (CompteCourant) cb;
Console.WriteLine(c3);
```

```
Titulaire : Mohamed
Solde : 1500 $CAD
```

En revanche l'opération suivante n'est pas correcte lors de l'exécution. Car l'objet cb contient un Compte Épargne et on veut le caster vers un compte courant : Chose impossible! D'où l'exception **InvalidCastException**.

```
CompteBancaire cb = new CompteEpargne("Jaque", 25000, "$CAD", 13);
CompteCourant c4 = (CompteCourant)cb;
Console.WriteLine(c4);
```



2. Opérateur AS

L'opérateur **as** convertit explicitement le résultat d'une expression en un type de valeur de référence ou nullable. Si la conversion n'est pas possible, l'opérateur as retourne **null**. Contrairement à une expression de cast, l'opératuer as ne lève jamais d'exception.

```
CompteBancaire cb1 = new CompteCourant("Mohamed", 1500, "$CAD",
"123456789", 1000);
CompteCourant c3 = cb1 as CompteCourant;
Console.WriteLine(c3);
```

```
Titulaire : Mohamed
Solde : 1500 $CAD
```

```
CompteBancaire cb2 = new CompteEpargne("Jaque", 25000, "$CAD", 13);
CompteCourant c4 = cb2 as CompteCourant;
Console.WriteLine(c4 == null);
```

```
True
```

IV. Exercice :

Faire le laboratoire sur les classes abstraites.