**-1. 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.

**2. Objet = état + actions**

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

* des informations (ou données) qui le caractérisent ;
* des actions (ou traitements) que l'on peut exercer sur lui ;

Imaginons que l'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.



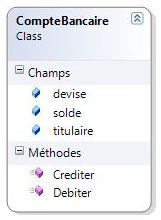
**3. La notion de classe**

 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.

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.

**4. Représentation graphique**

****

**II-C. Programmer avec des objets**

en C# de la classe CompteBancaire.

public class CompteBancaire

{

public string titulaire;

public double solde;

public string devise;

public void crediter(double montant)

{

solde = solde + montant;

}

public void debiter(double montant)

{

solde = solde - montant;

}

**II-C-2. Utilisation d'une 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 C# ci-dessous utilise la classe CompteBancaire définie plus haut pour créer le compte de Jean et y effectuer deux opérations.

static void Main(string[] args)

{

CompteBancaire comptePierre; // déclaration d'un nouvel objet

comptePierre = new CompteBancaire(); // instanciation de cet objet

// affectations de valeurs aux attributs

comptePierre.titulaire = "Pierre";

comptePierre.solde = 0;

comptePierre.devise = "euros";

// appels des méthodes

comptePierre.Crediter(300);

comptePierre.Debiter(500);

string description = "Le solde du compte de " + comptePierre.titulaire +

" est de " + comptePierre.solde + " " + comptePierre.devise;

Console.WriteLine(description);

}

À la fin du programme, l'attribut solde de l'objet comptePierre contient la valeur -200.

**II-C-3. Déclaration et instanciation**

On remarque que la création de l'objet comptePierre se fait en deux étapes :

1. Déclaration de l'objet ;
2. Instanciation de l'objet :
3. CompteBancaire comptePierre; // déclaration d'un nouvel objet
4. comptePierre = new CompteBancaire(); // instanciation de cet objet
5. // ...

* instancier, c'est créer un nouvel objet (opérateur new) ;
* initialiser, c'est donner une valeur initiale à quelque chose (opérateur =).

**II-C-4. Ajout d'une méthode**

Comme il est fastidieux de construire la description d'un compte bancaire à chaque fois que l'on en a besoin, on voudrait ajouter une opération de description du compte. Cette action est réalisée par une nouvelle méthode nommée Decrire ajoutée à la classe CompteBancaire. La description est renvoyée sous la forme d'une chaîne de caractères (string).

public class CompteBancaire

{

// ...

// Renvoie la description d'un compte

public string Decrire()

{

string description = "Le solde du compte de " + titulaire + " est de " + solde + " " + devise;

return description;

}

}

**II-C-5. Instanciation de plusieurs objets**

Voici un autre programme qui gère les comptes de Pierre et de Paul. Il permet à l'utilisateur de saisir un montant qui sera débité à Pierre et crédité à Paul.

static void Main(string[] args)

{

CompteBancaire comptePierre = new CompteBancaire();

comptePierre.titulaire = "Pierre";

comptePierre.solde = 500;

comptePierre.devise = "euros";

CompteBancaire comptePaul = new CompteBancaire();

comptePaul.titulaire = "Paul";

comptePaul.solde = 150;

comptePaul.devise = "euros";

Console.Write("Entrez le montant du transfert : ");

double montantTransfert = Convert.ToDouble(Console.ReadLine());

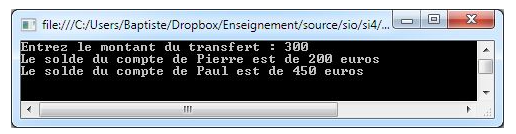
comptePierre.Debiter(montantTransfert);

comptePaul.Crediter(montantTransfert);

Console.WriteLine(comptePierre.Decrire());

Console.WriteLine(comptePaul.Decrire());

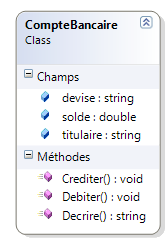
Resultat à la console :



**III. Principaux concepts objets**

L'objectif est de présenter les concepts essentiels de la programmation orientée objet.

Reprenons l'exemple de la classe CompteBancaire du chapitre précédent.



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.

public class CompteBancaire

{

public string titulaire;

public double solde;

public string devise;

// Constructeur

public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)

{

titulaire = leTitulaire;

solde = soldeInitial;

devise = laDevise;

}

// ...

**III-A-1. Rôle**

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

// déclaration et instanciation d'un nouvel objet en utilisant son constructeur

CompteBancaire comptePierre = new CompteBancaire("Pierre", 0, "euros");

// appels de méthodes

comptePierre.Crediter(300);

comptePierre.Debiter(500);

Console.WriteLine(comptePierre.Decrire());

**III-A-2. Constructeur par défaut**

Lorsqu'une classe ne définit aucun constructeur (comme dans l'exemple du chapitre précédent), 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.

**III-B. 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.

public class CompteBancaire

{

private string titulaire; // attribut privé

public double solde;

private string devise; // attribut privé

public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)

{

titulaire = leTitulaire;

solde = soldeInitial;

devise = laDevise;

}

// ...

À 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 générera une erreur lors de la compilation.

CompteBancaire comptePierre = new CompteBancaire("Pierre", 0, "euros");

comptePierre.titulaire = "Pierre"; // Erreur : titulaire est un attribut privé

comptePierre.solde = 500; // OK : solde est un attribut public

comptePierre.devise = "euros"; // Erreur : devise est un attribut privé

**III-B-2. 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.

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.

public class CompteBancaire

{

private string titulaire;

private double solde;

private string devise;

// ...

**III-B-3. 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).

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.

**III-C-1. Définition**

DÉFINITION : 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.

**III-C-2. Spécificités du langage C#**

En C#, les accesseurs prennent la forme de propriétés. Une propriété se manipule comme un champ, mais il s'agit en réalité d'un couple d'accesseurs get et set. Dans la plupart des autres langages, les accesseurs sont des méthodes de la forme getXXX et setXXX.

Voici la classe CompteBancaire modifiée pour intégrer des accesseurs vers ses attributs, ainsi que son nouveau diagramme de classe.

public class CompteBancaire

{

private string titulaire;

private double solde;

private string devise;

public string Titulaire

{

get { return titulaire; }

set { titulaire = value; }

}

public double Solde

{

get { return solde; }

set { solde = value; }

}

public string Devise

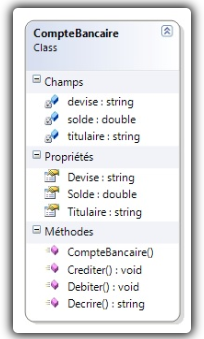
{

get { return devise; }

set { devise = value; }

}

// ...



on peut définir les propriétés de manière automatique. Dans ce cas, l'attribut disparaît de la définition de la classe : sa présence est implicite. Voici l'exemple précédent réécrit en utilisant des propriétés automatiques.

 public class CompteBancaire

{

// Définition des propriétés

public string Titulaire { get; set; }

public double Solde { get; set; }

public string Devise { get; set; }

// ...

**III-C-3. Avantages**

Voici un exemple d'utilisation des accesseurs définis précédemment.

CompteBancaire comptePierre = new CompteBancaire("Pierre", 1000, "euros");

// utilisation des propriétés (accesseurs)

string titulaireCompte = comptePierre.Titulaire; // en lecture (getter)

comptePierre.Solde = 500; // en écriture (setter)

comptePierre.Devise = "dollars"; // en écriture (setter)

En remplaçant l'accès direct à un attribut par l'utilisation d'une méthode, un accesseur permet d'effectuer des contrôles supplémentaires : respect d'une plage de valeurs, accès en lecture uniquement, etc.

**VI. La relation d'héritage**

L'héritage est l'un des mécanismes fondamentaux de la POO. Il permet de créer des classes à partir de classes existantes. L'objectif de ce chapitre est de découvrir son fonctionnement.

**VI-A-1. Exemple d'utilisation**

Reprenons la classe CompteBancaire utilisée dans un précédent chapitre.

// Définit un compte bancaire

public class CompteBancaire

{

private string titulaire; // Titulaire du compte

private double solde; // Solde du compte

private string devise; // Devise du compte

public string Titulaire

{

get { return titulaire; }

}

public double Solde

{

get { return solde; }

}

public string Devise

{

get { return devise; }

}

// Constructeur

public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)

{

titulaire = leTitulaire;

solde = soldeInitial;

devise = laDevise;

}

// Ajoute un montant au compte

public void Crediter(double montant)

{

solde = solde + montant;

}

// Retire un montant au compte

public void Debiter(double montant)

{

solde = solde - montant;

}

// Renvoie la description du compte

public string Decrire()

{

string description = "Le solde du compte de " + titulaire + " est de " + solde + " " + devise;

return description;

}

}

Supposons maintenant que nous ayons à gérer un nouveau type de compte : le compte épargne. Comme un compte classique, un compte épargne possède un titulaire, un solde et une devise. Sa spécificité est qu'il permet d'appliquer des intérêts à l'argent déposé sur le compte.

Bien sûr, il serait possible de concevoir une classe CompteEpargne totalement distincte de la classe CompteBancaire. Cependant, on constate qu'un compte épargne possède toutes les caractéristiques d'un compte bancaire, plus des caractéristiques spécifiques. Nous allons donc définir un compte épargne par héritage de la définition d'un compte bancaire.

public class CompteEpargne : CompteBancaire

{

private double tauxInteret;

public CompteEpargne(string leTitulaire, double soldeInitial, string laDevise, double leTauxInteret)

: base(leTitulaire, soldeInitial, laDevise)

// appel du constructeur de la classe CompteBancaire

// le mot-clé "base" permet d'accéder à la classe parente

{

tauxInteret = leTauxInteret;

}

// Calcule et ajoute les intérêts au solde du compte

public void AjouterInterets()

{

// ... (détaillé plus bas)

}

}

Dans la déclaration class CompteEpargne : CompteBancaire, les deux points spécifient que la classe CompteEpargne hérite de la classe CompteBancaire.

**Représentation graphique**

Le formalisme graphique UML décrit la relation d'héritage entre deux classes par une flèche pleine allant de la classe dérivée à la classe de base (les propriétés C# de CompteBancaire ont été masquées afin d'alléger le diagramme).

**VI-A-3. Définition**

DÉFINITION : l'héritage est un mécanisme objet qui consiste à définir une classe à partir d'une classe existante. Une classe héritant d'une autre, possède les caractéristiques de la classe initiale et peut définir ses propres éléments.

ATTENTION : le constructeur d'une classe dérivée doit obligatoirement faire explicitement appel au constructeur de la classe mère lorsque celui-ci prend des paramètres. C'est le cas dans notre exemple.

**VI-A-4. Avantages**

Grâce à la relation d'héritage, un objet de la classe CompteEpargne peut utiliser les fonctionnalités de la classe CompteBancaire sans avoir à les redéfinir. On peut donc débiter ou créditer un compte épargne exactement comme un compte bancaire.

double tauxInteret = 0.05; // taux d'intérêt : 5%

CompteEpargne comptePaul = new CompteEpargne("paul", 100, "dollars", tauxInteret);

// appel des méthodes de CompteBancaire sur le compte épargne

comptePaul.Debiter(1000);

comptePaul.Crediter(1500);

Console.WriteLine(comptePaul.Decrire()); // Affiche 600 $

Par contre, le calcul des intérêts (méthode AjouterInterets) ne peut se faire que sur un objet de la classe CompteEpargne. L'héritage est une relation unidirectionnelle.

// OK : comptePaul est un compte épargne

comptePaul.AjouterInterets();

Console.WriteLine(comptePaul.Decrire());

CompteBancaire comptePierre = new CompteBancaire("pierre", 100, "dollars");

// Erreur : comptePierre est un compte bancaire, pas un compte épargne

comptePierre.AjouterInterets();

Grâce à l'héritage, il est possible de réutiliser les fonctionnalités d'une classe existante en la spécialisant. Il est également possible de spécialiser une classe dérivée.

On voit bien tous les avantages que l'héritage peut apporter : gain de temps de développement, amélioration de la qualité du code, création de hiérarchies de classes reflétant précisément le domaine d'étude, etc.

**VI-A-5. Héritage et encapsulation**

Nous avons volontairement laissé de côté un point délicat. La méthode AjouterInterets de la classe CompteEpargne, qui doit ajouter les intérêts au solde, n'est pas encore définie. En voici une première version.

public class CompteEpargne : CompteBancaire

{

// ...

public void AjouterInterets()

{

// calcul des intérêts sur le solde

double interets = solde \* tauxInteret;

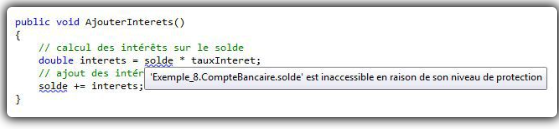
// ajout des intérêts au solde

solde += interets;

}

}

Cependant, le compilateur nous signale l'erreur suivante.



Dans cet exemple, la classe dérivée CompteEpargne tente d'accéder à l'attribut solde qui appartient à la classe de base CompteBancaire. Cependant, cet attribut est défini avec le niveau de visibilité private ! Cela signifie qu'il n'est utilisable que dans la classe où il est défini, et non dans les classes dérivées.

Pour interdire l'accès à un membre d'une classe (attribut, propriété C# ou méthode) depuis l'extérieur, tout en permettant son utilisation par une classe dérivée, il faut associer à ce membre un niveau de visibilité intermédiaire : protected.

public class CompteBancaire

{

private string titulaire;

protected double solde; // attribut protégé

private string devise;

// ...

Une autre solution à ce problème consiste à laisser le champ solde privé et à définir un accesseur protégé pour modifier le solde depuis les méthodes de la classe CompteEpargne.

public class CompteBancaire

{

private string titulaire;

private double solde;

private string devise;

// ...

public double Solde

{

get { return solde; } // accesseur public pour la lecture

protected set { solde = value; } // mutateur protégé pour la modification

}

// public string Solde { get; protected set; } // Equivalent avec une propriété automatique

// ...

Bien entendu, il faut alors utiliser le mutateur Solde et non plus l'attribut solde pour accéder au solde depuis la classe dérivée.

public class CompteEpargne : CompteBancaire

{

// ...

public void AjouterInterets()

{

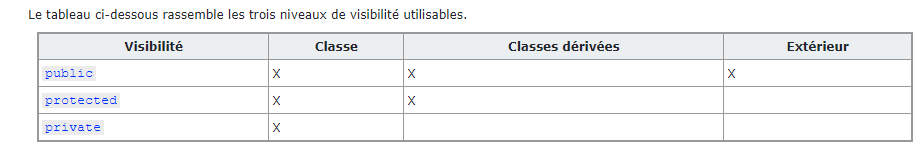
// utilisation du mutateur Solde pour accéder au solde du compte

double interets = Solde \* tauxInteret;

Solde += interets;

}

}



**VI-B. Polymorphisme**

Afin de gagner en généricité, on peut appliquer le même code à des objets de types différents, lorsque les classes de ces objets sont liées par héritage. C'est le principe du polymorphisme.

Prenons l'exemple d'une liste de comptes bancaires dont l'un est un compte épargne.

CompteBancaire compte1 = new CompteBancaire("Pierre", 300, "euros");

CompteEpargne compte2 = new CompteEpargne("Paul", 200, "dollars", 0.05);

CompteBancaire compte3 = new CompteBancaire("Jacques", 50, "euros");

List<CompteBancaire> listeComptes = new List<CompteBancaire>();

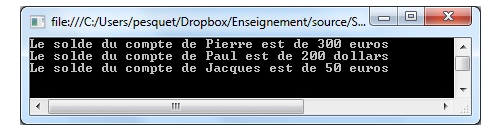
listeComptes.Add(compte1);

listeComptes.Add(compte2); // Est-ce bien un compte bancaire ?

listeComptes.Add(compte3);

foreach (CompteBancaire compte in listeComptes)

Console.WriteLine(compte.Decrire());



Ce code fonctionne ! Un compte épargne « est un » compte bancaire. On peut donc stocker un compte épargne dans une liste de comptes bancaires. On peut même appeler la méthode Decrire sur chacun des éléments de la liste de comptes bancaires. C'est un exemple très simple de ce que l'on appelle le polymorphisme.

DÉFINITION : utiliser le polymorphisme consiste à écrire un code générique qui pourra s'appliquer à des objets appartenant à des classes différentes.

Le polymorphisme rend le code plus concis, plus élégant et plus sûr. C'est un mécanisme à utiliser dès que l'occasion se présente. Il peut être enrichi grâce aux mécanismes que nous allons découvrir maintenant.

**VI-C. Classes et méthodes abstraites**

Notre modélisation objet du domaine doit refléter ces évolutions. Jusqu'à présent, un compte épargne hérite de toutes les caractéristiques d'un compte bancaire et en ajoute d'autres (taux d'intérêt). Nous voyons apparaître des éléments spécifiques à un compte courant : numéro de CB, découvert maximal. Il serait maladroit d'ajouter ces attributs à la classe CompteBancaire, puisqu'ils seraient hérités par la classe CompteEpargne alors qu'ils ne la concernent pas.

#### VI-C-1. Évolution des besoins

#### La bonne solution est de placer dans la classe CompteBancaire les éléments communs à tous les types de comptes. Deux autres classes, CompteCourant et CompteEpargne, héritent de CompteBancaire afin d'intégrer ces éléments communs. Chaque classe dérivée contient ce qui est spécifique à chaque type de comptes. Le diagramme ci-dessous reflète cette modélisation.

#### 

#### Une instance de CompteCourant ou de CompteEpargne représente respectivement un compte courant ou un compte épargne. Ce sont des concepts concrets du domaine d'étude.

#### En revanche, que représenterait une instance de CompteBancaire ? 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.

#### Pour refléter cela, on définit la classe CompteBancaire comme étant abstraite.

public abstract class CompteBancaire

{

// ...

}

En C# (ainsi qu'en Java et en C++), le mot-clé abstract permet de préciser qu'une classe est abstraite. Dans un diagramme de classe UML, le nom d'une classe abstraite est écrit en italiques.

DÉFINITION : 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.

#### Par opposition aux classes abstraites, les classes instanciables sont parfois appelées classes concrètes.

**I-C-4. Modélisation du comportement**

#### Les constructeurs de chaque classe sont simples à définir : ils doivent initialiser leurs attributs. Les constructeurs de CompteCourant et CompteEpargne feront appel à celui de CompteBancaire afin d'initialiser les attributs communs.

#### Les opérations que l'on souhaite appliquer aux comptes sont :

#### le dépôt d'argent (crédit) ;

#### le retrait d'argent (débit) ;

#### la description du compte.

#### Le dépôt d'argent fonctionne de la même manière pour tous les types de comptes : le solde est simplement augmenté du montant. Ce n'est pas le cas du débit. Chaque type de compte peut être débité, mais de manière très différente :

#### un compte courant autorise un découvert maximal ;un compte épargne limite le montant du retrait par rapport au solde. Pour traiter la problématique du débit d'argent, il faudrait pouvoir déclarer une opération de débit dans la superclasse CompteBancaire et laisser les classes dérivées définir comment cette opération est effectuée. Il existe une technique pour obtenir ce résultat : la définition d'une méthode abstraite. On modifie la classe CompteBancaire pour rendre la méthode Debiter abstraite, en la faisant précéder du mot-clé abstract.

-C-public abstract class CompteBancaire

{

private string titulaire;

private double solde;

private string devise;

public CompteBancaire(string leTitulaire, double soldeInitial, string laDevise)

{

titulaire = leTitulaire;

solde = soldeInitial;

devise = laDevise;

}

public double Solde

{

get { return solde; }

protected set { solde = value; }

}

public string Devise

{

get { return devise; }

}

public string Titulaire

{

get { return titulaire; }

}

public void Crediter(double montant)

{

solde += montant;

}

// La méthode Debiter est maintenant abstraite

public abstract void Debiter(double montant);

public string Decrire()

{

return "Le solde du compte de " + titulaire + " est de " + solde + " " + devise;

}

}

#### On ajoute dans la classe CompteCourant la méthode Debiter requise, avec la même signature que celle de la superclasse.

public class CompteCourant : CompteBancaire

{

private string numeroCB;

private double decouvertMaxi;

// Constructeur

public CompteCourant(string leTitulaire, double soldeInitial, string laDevise, string numeroCB, double decouvertMaxi)

: base(leTitulaire, soldeInitial, laDevise) // appel au constructeur de CompteBancaire

{

this.numeroCB = numeroCB;

this.decouvertMaxi = decouvertMaxi;

}

// Redéfinition de la méthode Debiter

public override void Debiter(double montant)

{

// on n'effectue le débit que si le solde final reste supérieur au découvert

if (Solde - montant >= decouvertMaxi)

Solde -= montant;

}

}

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

Suivant le même principe, on complète la définition de la classe CompteEpargne pour préciser de quelle manière un compte épargne est débité.

public class CompteEpargne : CompteBancaire

{

// ...

// Redéfinition de la méthode Debiter

public override void Debiter(double montant)

{

// Le montant maximal d'un retrait est la moitié du solde actuel

if (montant <= Solde / 2)

Solde -= montant;

}

}

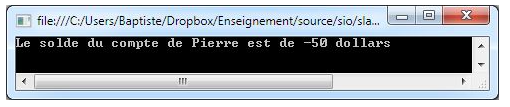
On peut maintenant instancier nos classes dérivées et tester leurs fonctionnalités.

CompteCourant compteCourant = new CompteCourant("Pierre", 250, "dollars", "1234 5678 9123 4567", -500);

compteCourant.Debiter(300);

compteCourant.Debiter(500);

Console.WriteLine(compteCourant.Decrire());



On constate que le second retrait de 500 dollars n'a pas eu lieu, puisqu'il aurait fait passer le solde en dessous du découvert maximal autorisé.

REMARQUE : le programme principal n'est pas informé de l'échec du second retrait, ce qui peut laisser croire que ce retrait a réussi. Nous découvrirons prochainement le mécanisme de remontée d'erreur que l'on utilise dans ces cas de figure.

CompteEpargne compteEpargne = new CompteEpargne("Paul", 1000, "euros", 0.04);

compteEpargne.Debiter(300);

compteEpargne.Debiter(500);

Console.WriteLine(compteEpargne.Decrire());

Une image contenant texte, capture d’écran, multimédia, Appareils électroniques

Description générée automatiquement

Ici encore, le second retrait n'a pas eu lieu : son montant est supérieur à la moitié du solde (700 euros au moment de l'appel).

**VI-C-7. Bilan**

DÉFINITION : une méthode abstraite (mot-clé abstract) déclare un comportement sans le définir. Elle doit être redéfinie (mot-clé override) dans toutes les classes dérivées.

Une classe comportant au moins une méthode abstraite est nécessairement une classe abstraite.

Déclarer une méthode abstraite dans une superclasse permet d'imposer à toutes les classes dérivées de fournir une implémentation de cette méthode. Ainsi, on demande à ces classes de fournir un certain comportement tout en les laissant choisir comment elles procèdent.

**VI-D. Méthodes virtuelles**

Intéressons-nous à la description d'un compte. Elle devrait renvoyer les données communes (solde, devise) et les données spécifiques au type (numéro de CB, découvert maximal ou taux d'intérêt), ce qui n'est pas le cas actuellement : seuls les attributs de la classe abstraite CompteBancaire sont affichés.

Dans ce cas de figure, on voudrait pouvoir utiliser le comportement commun (celui de CompteBancaire) et le compléter par un comportement particulier à chaque sous-classe. Pour cela, nous pouvons rendre la méthode Decrire virtuelle.

**VI-D-2. Mise en œuvre**

Modifiez la définition de Decrire dans CompteBancaire pour ajouter le mot virtual. Le reste de sa définition ne change pas.

public abstract class CompteBancaire

{

// ...

public virtual string Decrire()

{ // ... }

}

En faisant cela, on indique au compilateur que cette méthode est virtuelle, autrement dit susceptible d'être redéfinie dans une classe dérivée. C'est justement ce que nous allons faire dans CompteCourant et CompteEpargne pour intégrer à la description les données spécifiques à chaque type de comptes.

public class CompteCourant : CompteBancaire

{

// ...

// Redéfinition de la méthode Decrire

public override string Decrire()

{

return base.Decrire() + ". Son numéro CB est " + numeroCB +

" et son découvert maxi est de " + decouvertMaxi + " " + Devise + ".";

}

}

public class CompteEpargne : CompteBancaire

{

// ...

// Redéfinition de la méthode Decrire

public override string Decrire()

{

return base.Decrire() + ". Son taux d'intérêt est de " + (tauxInteret \* 100) + "%.";

}

}

Le mot-clé base permet d'accéder aux membres de la classe de base depuis une méthode d'une classe dérivée. Ici, base.Decrire() appelle la méthode Decrire de CompteBancaire.

Nous obtenons le résultat suivant.

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

À présent, chaque type de comptes dispose d'une description spécifique, et la description des informations communes à tous les comptes (titulaire, solde et devise) se fait sans aucune duplication de code.

**VI-D-3. Bilan**

DÉFINITION : une méthode virtuelle (virtual) fournit un comportement par défaut dans une classe. Elle peut être redéfinie (override) dans une classe dérivée.

Grâce aux méthodes virtuelles, on pousse encore plus loin les possibilités du polymorphisme.

ATTENTION : ne pas confondre méthode virtuelle et méthode abstraite :

* une méthode virtuelle définit un comportement, éventuellement redéfini ;
* une méthode abstraite déclare un comportement, obligatoirement redéfini.