PROGRAMMATION ORIENTEE OBJET

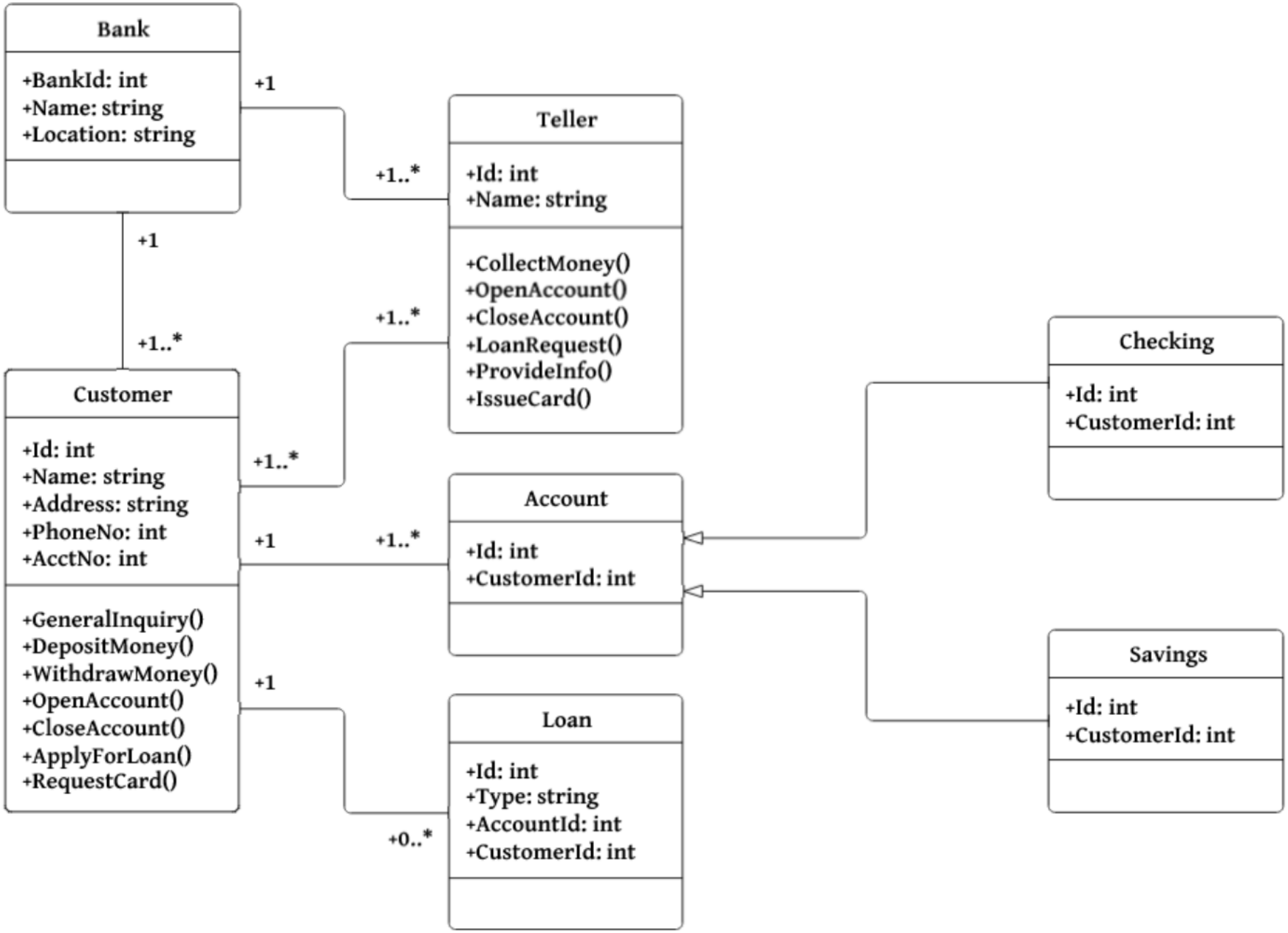
C#

INTRODUCTION

DEFINITION

- Nouveau PARADIGME pour FACILITER la PROGRAMMATION
- Concevoir applications sous forme de BRIQUES LOGICIELLES - OBJETS

- IDEE:
  - MODELISER COMMENT VA FONCTIONNER UNE APP => language UML
  - ⇒ CLASSES
  - ⇒ RELATIONS ENTRE CLASSES


Bank
+BankId: int
+Name: string
+Location: string
+1
+1..*
Teller
+Id: int
+Name: string
+CollectMoney()
+OpenAccount()
+CloseAccount()
+LoanRequest()
+ProvideInfo()
+IssueCard()
+1
+1..*
+1..*
Customer
+Id: int
+Name: string
+Address: string
+PhoneNo: int
+AcctNo: int
+GeneralInquiry()
+DepositMoney()
+WithdrawMoney()
+OpenAccount()
+CloseAccount()
+ApplyForLoan()
+RequestCard()
+1..*
+1
+1..*
Account
+Id: int
+CustomerId: int
+1
+0..*
Loan
+Id: int
+Type: string
+AccountId: int
+CustomerId: int
Checking
+Id: int
+CustomerId: int
Savings
+Id: int
+CustomerId: int

DEFINITION

- **Utilité :**
  - **MODELISATION claire**
  - **MAINTENANCE plus facile**
  - **REUTILISATION**

NOTION D’OBJET

NOTION D'OBJET

- Définition:
  - entité qui représente un élément concret

Objet = attributs + actions

- **Attributs = informations sur l'objet**
- **Actions = comportement de l'objet**

Exemple : **OBJET COMPTE BANCAIRE**

- **Quels sont ses attributs?**
- **Quelles sont les actions que l'on peut effectuer?**

- **ATTRIBUTS : titulaire, solde, devise**
- **ACTIONS : créditer, débiter**

NOTION DE CLASSE

NOTION DE CLASSE

- Définition:
  - **modèle** pour les objets (abstraction)
  - spécifie les informations et les actions qu'auront en commun tous les objets qui en sont issus
  - Créer une classe, c'est créer un type

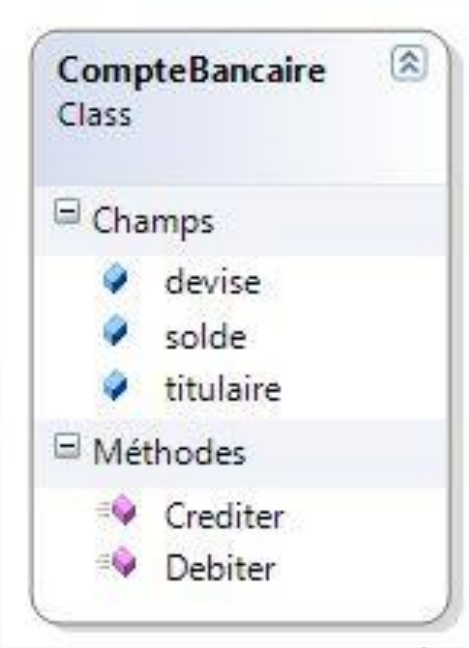
- **Différence CLASSE / OBJET:**
  - **Classe = modèle abstrait** Ex: classe CompteBancaire
  - **Objet = instance de classe** Ex: compte bancaire de Jean

NOTION DE CLASSE

- Représentation graphique:
  - **Unified Modelling Language (UML)**

  = permet de représenter les systèmes objets

- **Ex: CompteBancaire**
  - **Champs = attributs**
  - **Méthodes = actions**



NOTION DE CLASSE

- Ecriture d’une classe:
  - **Mot-clef : class + nom_classe{}**
  - Champs + Méthodes

Exemple : **ECRITURE CLASSE COMPTE BANCAIRE**

QUESTIONS A SE POSER

- **ATTRIBUTS**
  - Quels sont les attributs de la classe?
  - De quel type sont ces attributs?
- **ACTIONS**
  - Quelles sont les actions que l'on peut effectuer?
  - Les fonctions prennent-t-elles un/des paramètres?
  - Les fonctions doivent-elles renvoyer une information?

Exemple : **CLASSE COMPTE BANCAIRE**

```
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;
    }

// Renvoie la description d'un compte
    public string Decrire()
    {
        string description = "Le solde du compte de " +  titulaire + " est
de " + solde + " " + devise;
        return description;
    }
}
```

NOTION DE CLASSE

- Utilisation d'une classe:
  - Créer un **objet de type …**
  - Différents objets d'une même classe:
    - Même attributs et méthodes
    - Valeurs différentes

NOTION DE CLASSE

- ETAPES:
  - Déclaration de l'objet :

Nom_Classe + nom_variable_objet

  - Instanciation de l'objet : **new**
  - Initialisation de l'objet : **=**

Exemple : **UTILISATION CLASSE COMPTE BANCAIRE**

```
static void Main(string[] args)
{
    CompteBancaire comptePierre;           // déclaration d'un nouvel objet
    comptePierre = new CompteBancaire();   // instanciation de cet objet

    // initialisation des valeurs des 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);
}
```

Dans cet exemple, combien vaut le solde du compte de Pierre à al fin de l'exécution du programme?

Exemple : **UTILISATION CLASSE COMPTE BANCAIRE – PLUSIEURS OBJETS**

```
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());
}
```

NOTION DE CLASSE

- Classe SYSTEM.OBJECT:
  - **CLASSE DE BASE FONDAMENTALE**
    - toute classe C# a accès aux méthodes de base issues de la Classe Object
    - toute classe C# hérite directement ou indirectement de la Classe Object et peut utiliser ou redéfinir ses méthodes

- **Ex : redéfinition de la méthode .ToString()**

```
public class CompteBancaire
{
    public override string ToString()
    {
        return "Le solde du compte de " + titulaire + " est de " + solde + "
" + devise;
    }
```

PRINCIPAUX CONCEPTS OBJETS

CONSTRUCTEUR

CONSTRUCTEUR

- Définition:
  - **méthode spécifique** dont le rôle est de **construire un objet**
  - **BUT : initialiser attributs de l'objet créé**
    - Constructeur par défaut
    - Constructeur paramétrisé

DEFINITION CONSTRUCTEUR CLASSE COMPTE BANCAIRE

```
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;
    }
```

UTILISATION CONSTRUCTEUR CLASSE COMPTE BANCAIRE

```
// déclaration et instanciation d'un nouvel objet en utilisant son
constructeur
CompteBancaire comptePierre = new CompteBancaire("Pierre", 0, "euros");
```

ENCAPSULATION

ENCAPSULATION

- Définition:
  - Permet de **restreindre l'accès** à certains éléments d'une classe (le plus souvent ses attributs)
  - Objectif de l'encapsulation = ne laisser accessible que le strict nécessaire pour que la classe soit utilisable

ENCAPSULATION

MODIFICATEURS D’ACCES

ENCAPSULATION – MODIFICATEURS D'ACCES

- Mots-clés **public** et **private** pour déterminer le niveau de visibilité / accessibilité des éléments de la classe :
  - **public** => librement utilisable depuis le reste du programme
  - **privé** => uniquement utilisable depuis les méthodes de la classe elle-même
- BONNE PRATIQUE : marquer en ***private*** tous les attributs d'une classe afin d'assurer leur encapsulation par défaut.

Ex : créer un compte bancaire dont on ne puisse pas changer les attributs

```
public class CompteBancaire
{
    private string titulaire;   // attribut privé
    private double solde;       // attribut privé
    private string devise;      // attribut privé

    public CompteBancaire(string leTitulaire, double soldeInitial, string
laDevise)
    {
        titulaire = leTitulaire;
        solde = soldeInitial;
        devise = laDevise;
    }
```

CONSEQUENCE : seule manière de définir des valeurs pour *les attributs*

=> utiliser le constructeur

PROBLEME : IMPOSSIBLE D'ACCEDER AUX ATTRIBUTS EN DEHORS DE LA CLASSE

ENCAPSULATION

ACCESSEURS

ENCAPSULATION – ACCESSEURS

- Définition:
  - Méthode publique qui permet d'accéder à un attribut privé
  - 2 sortes:
    - **Getter** : lire valeur de l'attribut privé
    - **Setter** : modifier valeur de l'attribut privé

Ex : classe Compte Bancaire avec accesseurs

```
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; }
    }
```

RELATIONS D'ASSOCIATION

RELATIONS D'ASSOCIATION

- Définition:
  - OOP : chaque objet joue un rôle précis
  - OOP : communication entre les objets

  => Interactions entre les différents objets vont permettre à l'application de réaliser les fonctionnalités attendues

- IMPORTANT : Application évolutives – **Evolution des besoins**

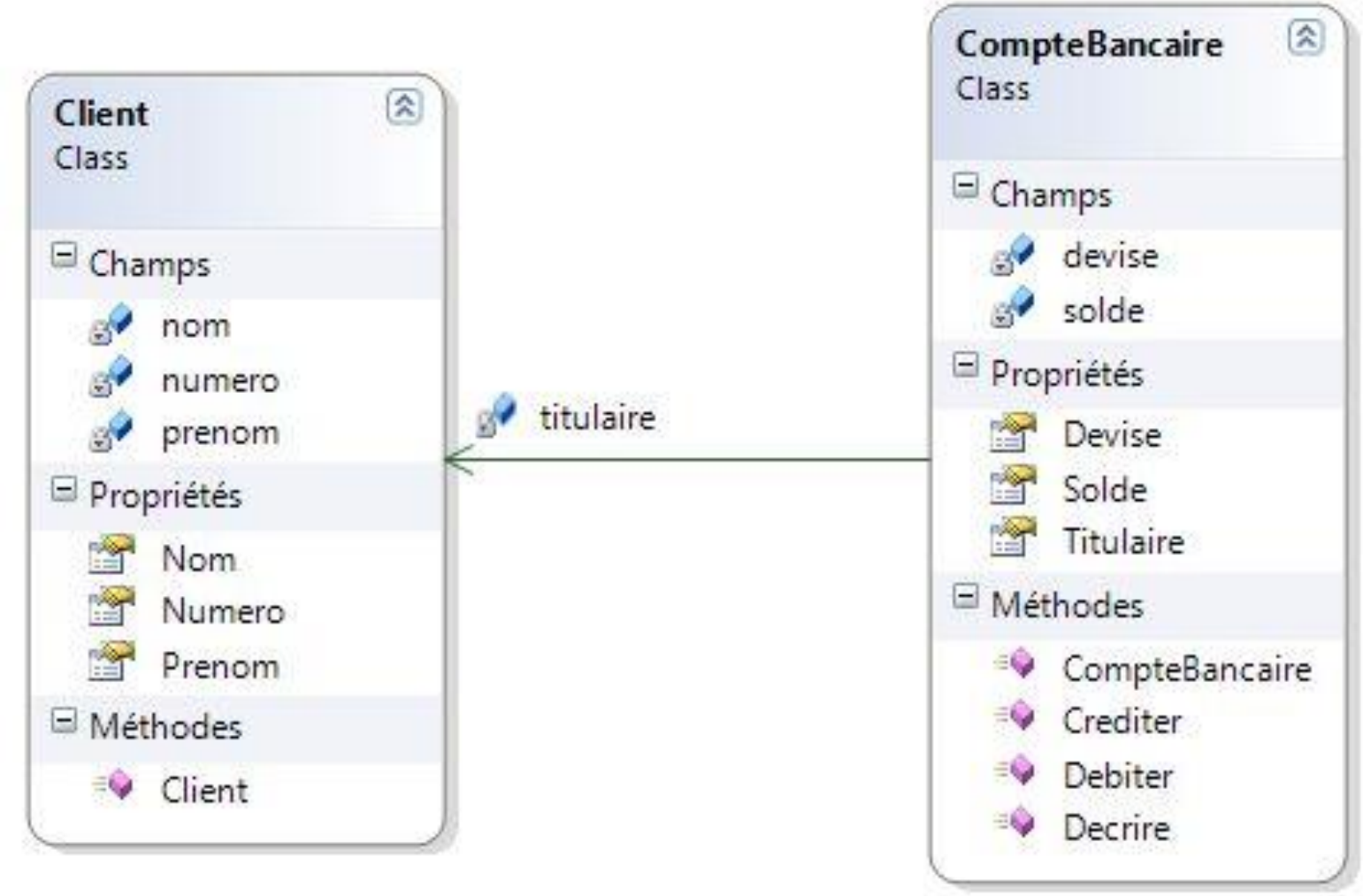
Ex : classe Compte Bancaire -

EVOLUTION = infos détaillées sur titulaire d'un compte

QUESTIONS A SE POSER

- Que modélise un compte bancaire?
- Est-il pertinent d'intégrer infos sur titulaire dans classe Compte Bancaire ?
- Si non, doit-on créer une autre classe ?
- Comment faire interagir les classes entre elles?

Ex : classe Compte Bancaire - Classe Client => relation « a un »

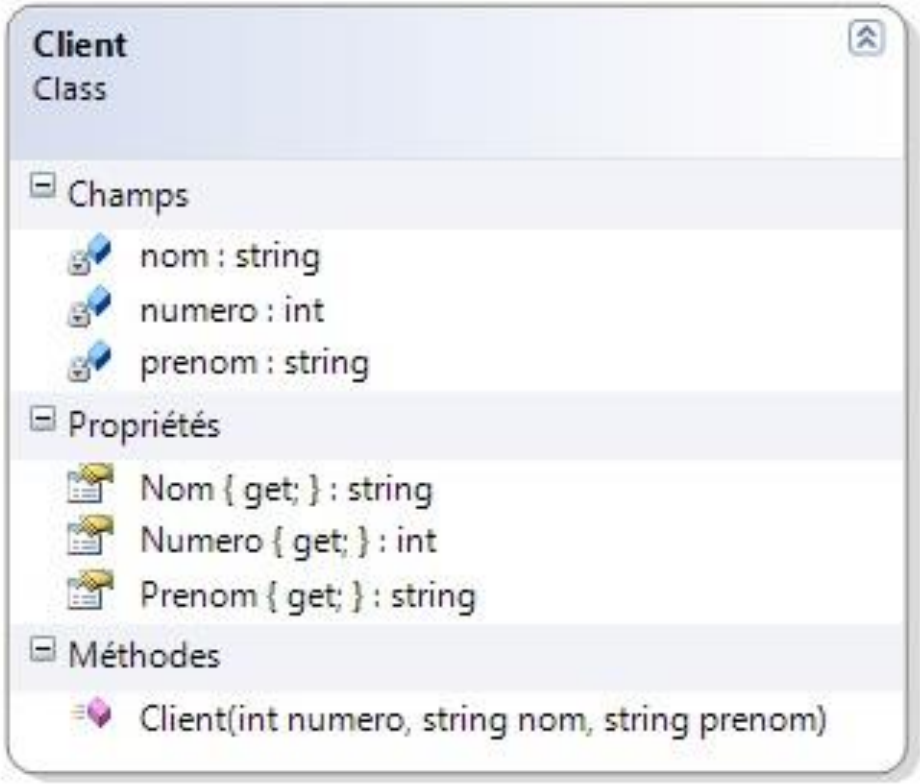


Ex : classe Compte Bancaire -

EVOLUTION = ETAPE 1 : création classe Client

QUESTIONS A SE POSER

- Quels sont les attributs d'un client?
- Quelles sont les méthodes possibles?



Ex : classe Compte Bancaire - EVOLUTION = création classe Client

```
// Modélise un client
public class Client
{
    private int numero;     // Numéro de compte
    private string nom;     // Nom
    private string prenom; // Prénom

    public int Numero
    {
        get { return numero; }
    }

    public string Nom
    {
        get { return nom; }
    }

    public string Prenom
    {
        get { return prenom; }
    }

    public Client(int leNumero, string leNom, string lePrenom)
    {
        numero = leNumero;
        nom = leNom;
        prenom = lePrenom;
    }
}
```

Ex : classe Compte Bancaire -

EVOLUTION = ETAPE 2 : modification de la classe Compte Bancaire

QUESTIONS A SE POSER

- Comment modéliser l'information « **ce compte a pour titulaire ce client** » ?
- Quelles sont les éléments à modifier dans la classe ?

  - **Eléments à modifier:**
    - **Type de l'attribut titulaire**
    - **Accesseurs**
    - **Constructeur**

Ex : classe Compte Bancaire -

EVOLUTION = modification classe Compte Bancaire

```
private Client titulaire;  // type string => type Client

    public Client Titulaire
    {
        get { return titulaire; }
    }

    public CompteBancaire(Client leTitulaire, double soldeInitial, string
laDevise)
    {
       titulaire = leTitulaire;
       solde = soldeInitial;
       devise = laDevise;
    }
```

Désormais, création d'un nouveau compte nécessite de passer en paramètre du constructeur le Client titulaire

POLYMORPHISME

POLYMORPHISME

- Définition:
  - écrire un code générique pouvant s'appliquer à des objets appartenant à des classes différentes mais liées par héritage

POLYMORPHISME

- Ex:
  - Méthode ToString()

HERITAGE

HERITAGE

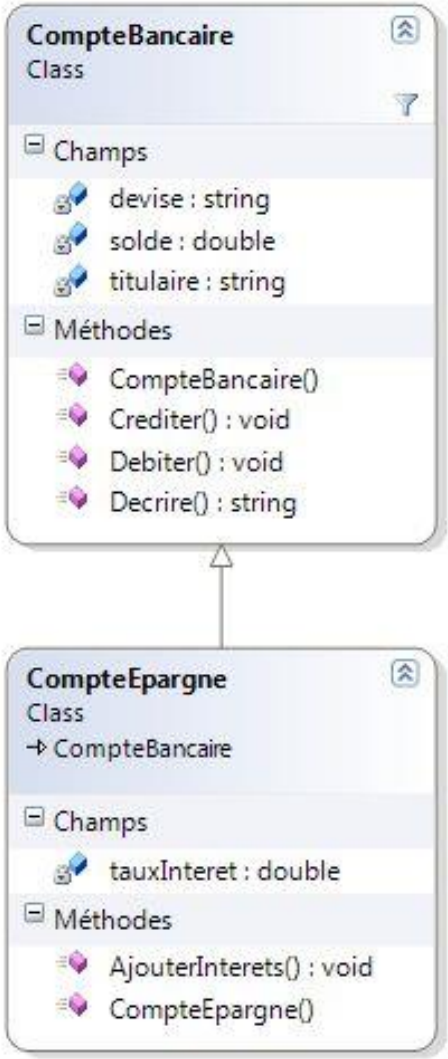
- Définition:
  - mécanisme qui consiste à définir une classe (classe enfant/classe dérivée) à partir d'une classe existante (classe parente)
    - **classe enfant :** possède les caractéristiques de la classe mère et peut définir ses propres éléments
    - **classe dérivée** : **spécialisation** de la classe parente
  - Relation de type « **est un** » entre les classes

- IMPORTANT : Application évolutives – **Evolution des besoins**

EVOLUTION = création du type de compte Comte Epargne

Ex : classe Compte Bancaire – Classe Compte Epargne

=> relation « est un »



Compte épargne =

- caractéristiques d'un compte classique : titulaire, solde, devise
- spécificité : appliquer des intérêts à l'argent déposé sur le compte

EVOLUTION = création du type de compte Comte Epargne

```
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()
    {
                // calcul des intérêts sur le solde
        double interets = solde * tauxInteret;
        // ajout des intérêts au solde
        solde += interets;
    }
}
```

HERITAGE

- AVANTAGES:
  - Utiliser fonctionnalités définies dans classe parente
    - **gain de temps de développement,**
    - **amélioration de la qualité du code,**
    - **création de hiérarchies de classes reflétant précisément le domaine d'étude, etc.**

PROBLEME = méthode AjouterInterets() tente d'accéder à attribut solde de la classe Compte Bancaire qui est en privé

```
public void AjouterInterets()
{
    // calcul des intérêts sur le solde
    double interets = solde * tauxInteret;
    // ajout des intér
    solde += interets;
}
```

'Exemple_8.CompteBancaire.solde' est inaccessible en raison de son niveau de protection

HERITAGE

- SOLUTION 1:
  - Jouer sur le niveau de visibilité

VISIBILITÉ	CLASSE	CLASSES DÉRIVÉES	EXTÉRIEUR
PUBLIC	X	X	X
PROTECTED	X	X	
PRIVATE	X		

HERITAGE

- SOLUTION 2:
  - Définir un **accesseur protégé** pour modifier le solde depuis les méthodes des classes dérivées

```
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
```

HERITAGE

- SOLUTION 2:
  - Utiliser ***S****olde* et non plus ***s****olde* 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;
    }
}
```

CLASSES et METHODES ABSTRAITES

CLASSES ABSTRAITES

- Définition:
  - définit un concept abstrait, incomplet ou théorique
  - rassemble des éléments communs à plusieurs classes dérivées.
  - pas destinée à être instanciée

EVOLUTION = compte bancaire:

- **soit un compte courant**
- **soit un compte épargne**

- **Abstraction** : un compte bancaire en général n'a pas d'existence concrète, c'est soit un compte courant, soit un compte épargne

- **Eléments concrets** : instance de CompteCourant ou instance de CompteEpargne représente respectivement un compte courant ou un compte épargne

EVOLUTION = compte bancaire:

- **soit un compte courant**
- **soit un compte épargne**

- QUELLES SONT LES CARACTERISTIQUES COMMUNES?
  - titulaire, solde et devise
- QUELLES SONT LES SPECIFICITES?

COMPTE COURANT	COMPTE EPARGNE
Numéro de carte bancaire	Taux d'intérêt
Découvert maximal	

EVOLUTION = compte bancaire:

- **soit un compte courant**
- **soit un compte épargne**

Solution :

- Placer dans la classe CompteBancaire les éléments communs à tous les types de comptes.
- Créer deux autres classes, CompteCourant et CompteEpargne, qui :
  - héritent de CompteBancaire afin d'intégrer ces éléments communs
  - contiennent chacune ce qui leur est spécifique

EVOLUTION = compte bancaire:

- soit un compte courant
- soit un compte épargne

```
public abstract class CompteBancaire
```

```
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;
    }
```

METHODES ABSTRAITES

- 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

- Le terme **implémenter** signifie : rendre concret, traduire en code.

- La **signature** d'une méthode est constituée de son nom et de la liste de ses paramètres.

METHODES ABSTRAITES

- **DECLARER** : déclarer une méthode, c'est dire ce que doit faire cette méthode, sans dire comment
- **DEFINIR / IMPLEMENTER** : définir une méthode, c'est dire comment cette méthode fait ce qu'elle doit faire, autrement dit l'implémenter.

EVOLUTION = méthode Décrire de la Classe Compte Bancaire

```
public abstract class CompteBancaire
{
    // 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;
    }
}
```

Rôle des classes qui héritent de CompteBancaire => fournir une implémentation respectant la signature de la méthode **sinon ne compilent plus !**

```
public class CompteCourant : CompteBancaire
{
    // 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; } }
```

CLASSES et METHODES VIRTUELLES

METHODES VIRTUELLES

- Définition:
  - une méthode VIRTUELLE (mot-clé **virtual**) fournit un comportement par défaut dans une classe
  - une méthode VIRTUELLE peut être redéfinie (mot-clé **override**) dans les classes dérivées

- ATTENTION : **ne pas confondre** méthode virtuelle et méthode abstraite :
  - Méthode virtuelle : définit un comportement, éventuellement redéfini.
  - Méthode abstraite : déclare un comportement, obligatoirement redéfini.

EVOLUTION = méthode Décrire de la Classe Compte Bancaire

```
public abstract class CompteBancaire
{
    // ...

    public virtual string Decrire()
    { // ... }
}
```

EVOLUTION = méthode Décrire classes Compte Courant & Compte Epargne

```
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) + "%.";
    }
}
```

SYNTAXE :

Mot-clé **base** permet d'accéder aux membres de la classe de base depuis une méthode d'une classe dérivée

=> base.Decrire() appelle la méthode Decrire de CompteBancaire.

MEMBRES DE CLASSE

ATTRIBUTS DE CLASSE

- Définition:
  - Attribut **lié** à la classe parente
  - Attribut **partagé** entre toutes les instances de la classe

Attribut de classe permet **d'*internaliser*** un attribut au sein de la classe parente

EVOLUTION = identifier les comptes bancaires par un numéro unique

SOLUTION =>

- Créer un attribut de classe **numeroProchainCompte** dans classe CompteBancaire = stocker l'information sur le prochain numéro de compte

=> à chaque appel du constructeur de CompteBancaire, la valeur courante de numeroProchainCompte récupérée et incrémentée.

EVOLUTION = identifier les comptes bancaires par un numéro unique

```
public class CompteBancaire
{
    // ...
    private int numero;
    private static int numeroProchainCompte = 1;  // Numéro du prochain
compte créé

    // ...
```

⇒ lier l'attribut à la classe CompteBancaire elle-même, et non à chacune de ses instances:

- ❑ même attribut serait partagé
- ❑ obtenir des numéros de compte uniques.

METHODES DE CLASSE

- Définition:
  - Méthode **liée** à la classe parente
  - Méthode **partagée** entre toutes les instances de la classe

=> définir un comportement indépendant de toute instance

REMARQUE : méthode de classe ≠ méthode d'instance => utilisation différente :

Méthode de classe = peut s'utiliser en l'absence de toute instance de la classe : liée à la classe et non à ses instances !

Ex:

- Console.WriteLine
- Convert.ToDouble.
- Program.Main

EVOLUTION = récupérer le numéro du prochain compte via une méthode

```
public static int GetNumeroProchainCompte()
{
        return numeroProchainCompte;
}
```

LIMITES interactions entre membres de classe et membres d'instance :

- une méthode d'instance peut utiliser un membre de classe (attribut ou méthode).
- une méthode de classe peut utiliser un membre de classe.
- une méthode de classe ne peut pas utiliser un membre d'instance (attribut ou méthode).
- une méthode de classe ne peut pas utiliser le mot-clé this.

EXCEPTIONS

EXCEPTIONS

- Définition:
  - PROBLEME qui apparaît pendant le déroulement d'un programme et qui empêche la poursuite normale de son exécution.
  - DANGER ! Une exception non interceptée provoque un arrêt brutal de l'exécution d'un programme.

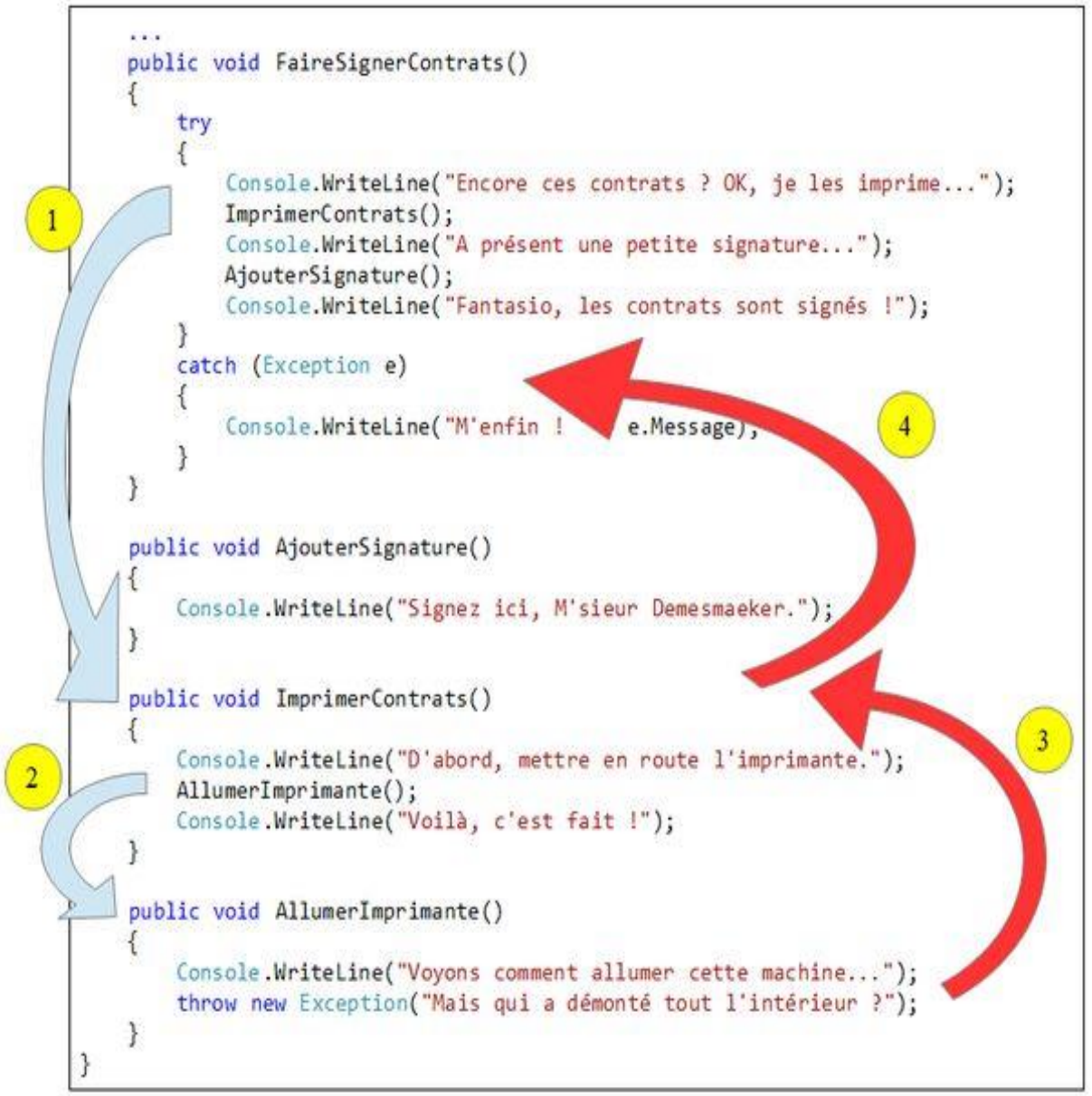
EXCEPTIONS

Syntaxe générale de la gestion des exceptions est la suivante :

```
try
{
    // code susceptible de lever des exceptions
}
catch (Exception e)
{
    // code de gestion de l'exception qui s'est produite dans le bloc
        Console.WriteLine("M'enfin ! " + e.Message);
        throw new Exception("Beaucoup trop de lettres..."); //renvoie
nouvelle exception
}
```

- SYNTAXE
  - **try** délimite un bloc de code dans lequel des exceptions peuvent se produire.
  - **catch** délimite un bloc de code qui intercepte et gère les exceptions levées dans le bloc try associé.
  - **throw** lève une nouvelle exception.

EXCEPTIONS



- REMARQUE : une exception est un objet : Dans bloc catch.

- exception est instanciée comme un objet classique grâce au mot-clé **new**

- puis exception levée/ jetée avec mot-clé **throw**

- variable **e** est un objet, instance de la classe Exception, avec une propriété Message, qui renvoie le message véhiculé par l'exception

Exception remonte la chaîne des appels dans l'ordre inverse, jusqu'à être interceptée dans un bloc **catch**