

# EXERCICES PROGRAMMATION ORIENTÉ OBJET (POO)

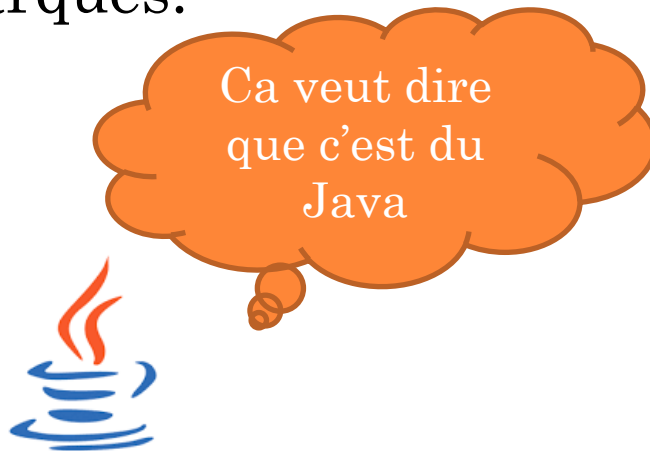
A faire en Java ou C #? A voir avec le formateur!

1

# EXERCICES

- Compte Bancaire
- Point
- Fraction
- Banque
- Jeu 421

Remarques:



C#





# COMPTE BANCAIRE

Notions d'attributs, de constructeurs, de  
réutilisation de méthodes

3

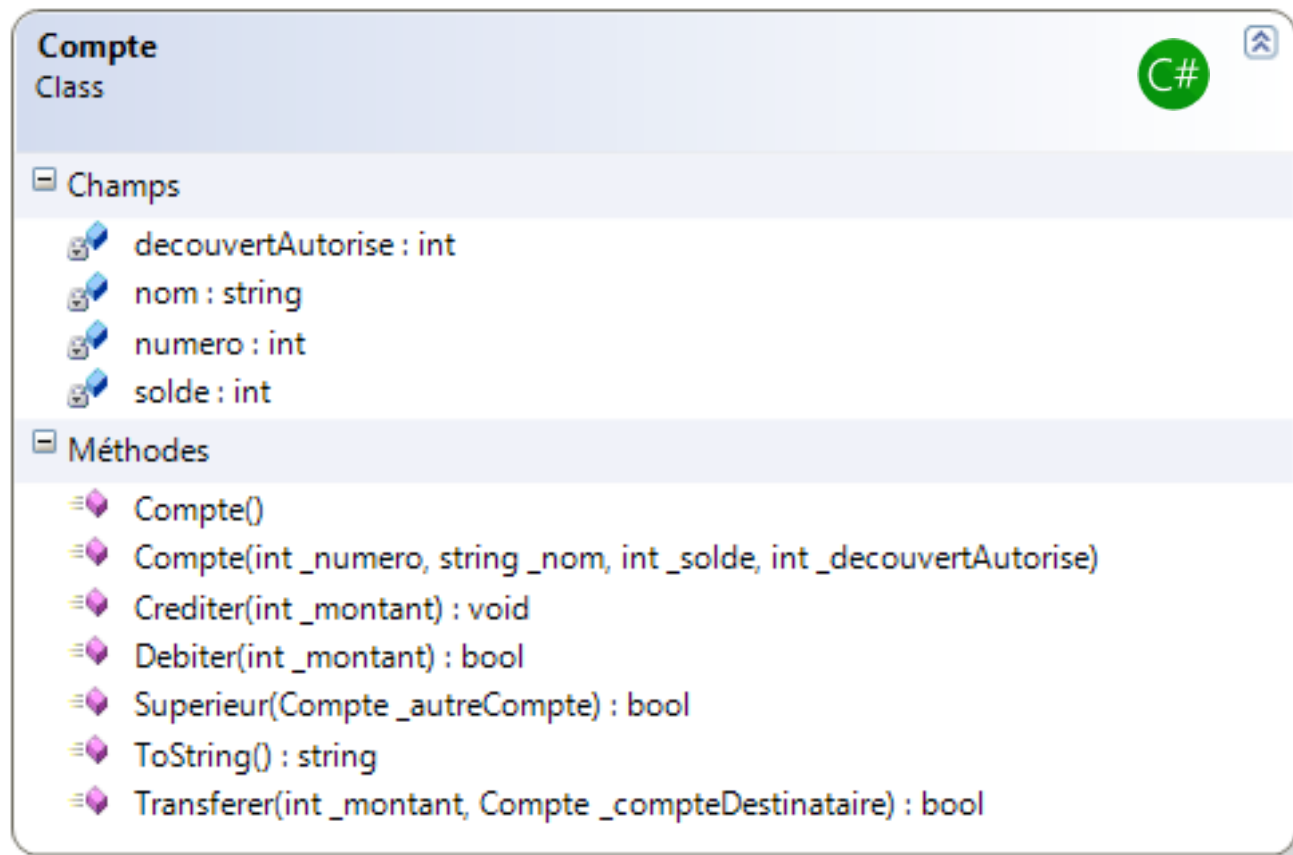
# COMPTE BANCAIRE

- Un compte bancaire (simplifié) est caractérisé par :
  - un numéro unique
  - le nom de son propriétaire
  - son solde (montant restant sur le compte): il peut être négatif
  - le montant du découvert autorisé (chiffre négatif) : le solde ne peut descendre en dessous.
- Nous donnons à cette classe les comportements
  - Donner une représentation textuelle de toutes ses informations
  - Créditer d'un montant fourni
  - Débiter le solde d'un montant fourni, mais attention un "drapeau" (booléen) indiquera si l'opération a pu se réaliser
  - Transférer un montant, du compte courant vers un autre compte; même remarque que pour le paragraphe précédent.
  - Comparer le solde de l'objet courant avec le solde d'un autre compte fourni, le résultat sera un booléen

1. Dessiner le diagramme de classe Compte sur papier

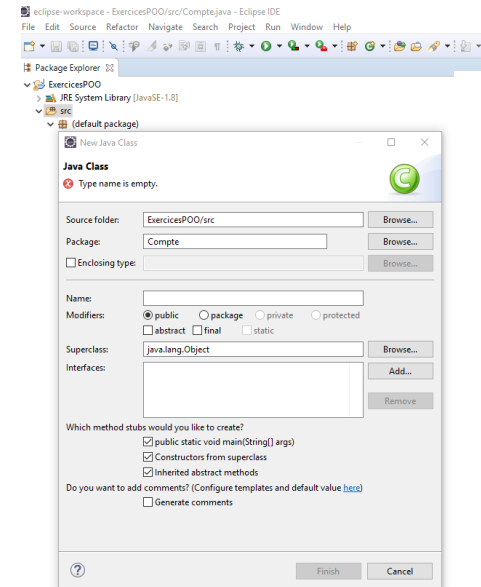
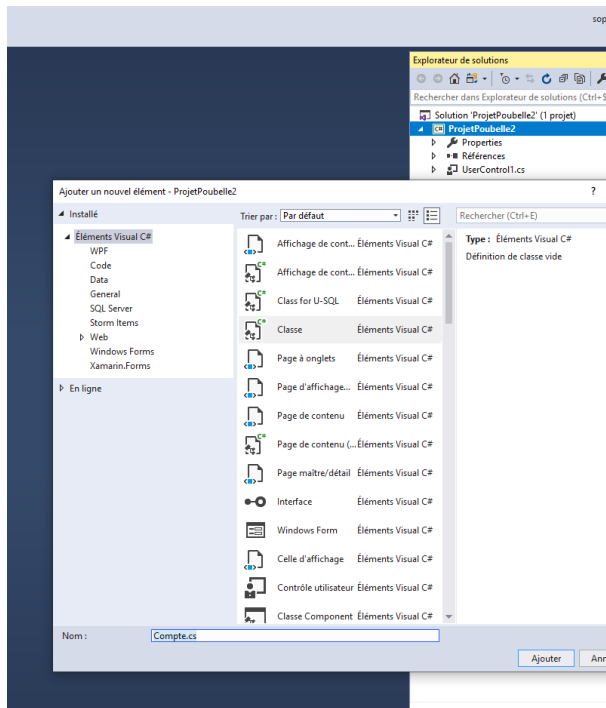
# COMPTE BANCAIRE

## ○ Correction



# COMPTE BANCAIRE- AJOUTER UNE CLASSE

- Une classe correspond toujours a un fichier quand on travaille proprement:
  - Compte est définit dans le fichier Compte.cs en C#
  - Compte est définit dans le fichier Compte.java en Java



# COMPTE BANCAIRE – LES CONSTRUCTEURS

- Implémentez les constructeurs et la méthode ToString() en C# ou toString() en Java afin de réaliser l'exemple suivant

```
class Program
```



```
{  
    static void Main()  
    {  
        Compte c = new Compte();  
        Compte c1 = new Compte(12345, "toto", 1000, -500);  
        Console.WriteLine(c.ToString());  
        Console.WriteLine(c1.ToString());  
    }  
}
```

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    Compte c = new Compte();  
    Compte c1 = new Compte(12345, "toto", 1000, -500);  
    System.out.println(c);  
    System.out.println(c1);  
}
```



# COMPTE BANCAIRE – LES CONSTRUCTEURS

- Voici une proposition d'implémentations des toString

```
public override string ToString()
{
    return "numéro :"+numero+" nom :"+nom+" solde :"+solde+" découvert autorisé:"+decouvertAutorise;
}
```

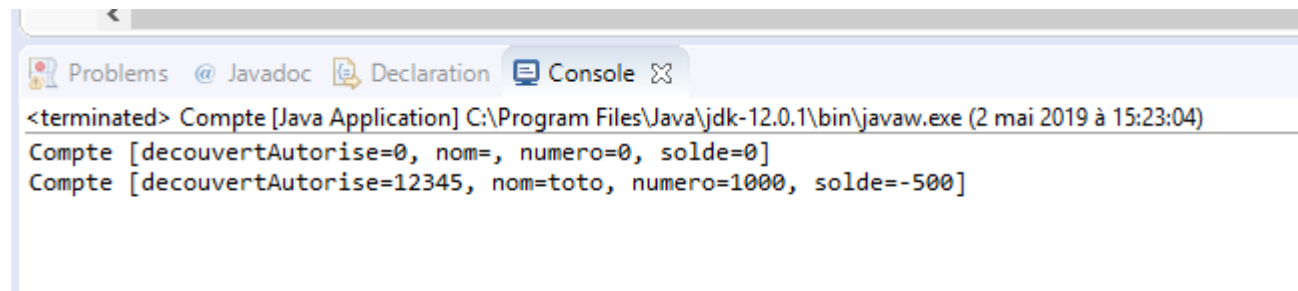


```
@Override
public String toString() {
    return "Compte [decouvertAutorise=" + decouvertAutorise + ", nom=" + nom + ", numero=" + numero + ", solde="
        + solde + "];"
}
```



- Le résultat de l'exécution doit être le suivant

```
numéro : 0 nom : solde : 0 découvert autorisé : 0
numéro :12345 nom :toto solde :1000 découvert autorisé :-500
```





# COMPTE BANCAIRE - MÉTHODES

- Implémentez les méthodes Crediter et Debiter afin de réaliser l'exemple suivant



```
Compte b = new Compte(545454, "Laurent", 2000, -500);
b.Crediter(100);
Console.WriteLine(b);
bool ok = b.Debiter(100000);
Console.WriteLine(b.ToString());
if (ok)
{
    Console.WriteLine("Débit réussi!");
}
else
{
    Console.WriteLine("Débit pas réussi!");
}
```



```
Compte b = new Compte(545454, "Laurent", 2000, -500);
b.crediter(100);
System.out.println(b); //raccourci sysout + Ctrl+ espace
Boolean ok = b.debiter(100000);
System.out.println(b.toString());
if (ok)
{
    System.out.println("Débit réussi!");
}
else
{
    System.out.println("Débit pas réussi!");
}
System.out.println();
```

# COMPTE BANCAIRE - MÉTHODES

- Le résultat de l'exécution doit être le suivant

```
numéro :545454 nom :Laurent solde 2100 autorisé-500  
numéro :545454 nom :Laurent solde 2100 autorisé-500  
Débit pas réussi!
```



```
Compte [decouvertAutorise=545454, nom=Laurent, numero=2000, solde=-400]  
Compte [decouvertAutorise=545454, nom=Laurent, numero=2000, solde=-400]  
Débit pas réussi!
```



# COMPTE BANCAIRE – MÉTHODE TRANSFÉRER

- Implémentez les méthodes afin de réaliser l'exemple suivant

```
Compte c1 = new Compte(12345, "toto", 1000, -500);  
Compte c2 = new Compte(45657, "titi", 2000, -1000);  
c1.Transferer(1300, c2);  
Console.WriteLine(c1.ToString());  
Console.WriteLine(c2.ToString());
```

C#

- Le résultat de l'exécution doit être le suivant

```
numéro :12345 nom :toto solde :-300 découvert autorisé :-500  
numéro :45657 nom :titi solde :3300 découvert autorisé :-1000
```

C#

# COMPTE BANCAIRE – MÉTHODE TRANSFÉRER

- Implémentez les méthodes afin de réaliser l'exemple suivant

```
Compte c1 = new Compte(12345, "toto", 1000, -500);  
Compte c2 = new Compte(45657, "titi", 2000, -1000);  
c1.Transferer(1300, c2);  
Console.WriteLine(c1.ToString());  
Console.WriteLine(c2.ToString());
```



```
Compte c1 = new Compte(12345, "toto", 1000, -500);  
Compte c2 = new Compte(45657, "titi", 2000, -1000);  
c1.transferer(1300, c2);  
System.out.println(c1.toString());  
System.out.println(c2.toString());
```



# COMPTE BANCAIRE - MÉTHODE TRANSFÉRER

- Et le résultat doit être

```
numéro :12345 nom :toto solde :1000 découvert autorisé :-500  
numéro :45657 nom :titi solde :2000 découvert autorisé :-1000
```



```
Compte [decouvertAutorise=12345, nom=toto, numero=1000, solde=-500]  
Compte [decouvertAutorise=45657, nom=titi, numero=2000, solde=-1000]
```



# COMPTE BANCAIRE –MÉTHODE SUPÉRIEUR

- Implémentez les méthodes afin de réaliser l'exemple suivant



C#

```
Compte c1 = new Compte(12345, "toto", 1000, -500);
Compte c2 = new Compte(45657, "titi", 2000, -1000);
if (c1.Superieur(c2))
{
    Console.WriteLine("superieur");
}
else
{
    Console.WriteLine("inférieur");
}
```

```
Compte c1 = new Compte(12345,"toto",1000,-500);
Compte c2 = new Compte(45657,"titi",2000,-1000);
if (c1.superieur(c2)) {
    System.out.println("superieur");
}
else
{
    System.out.println("inferieur");
}
```

- Le résultat de l'exécution doit afficher inférieur

- Valider au près de votre formateur si il est présent!



17

## POINT

Notions de surcharge de constructeurs, et de différentiation entre objet issu de la même classe.



# POINT

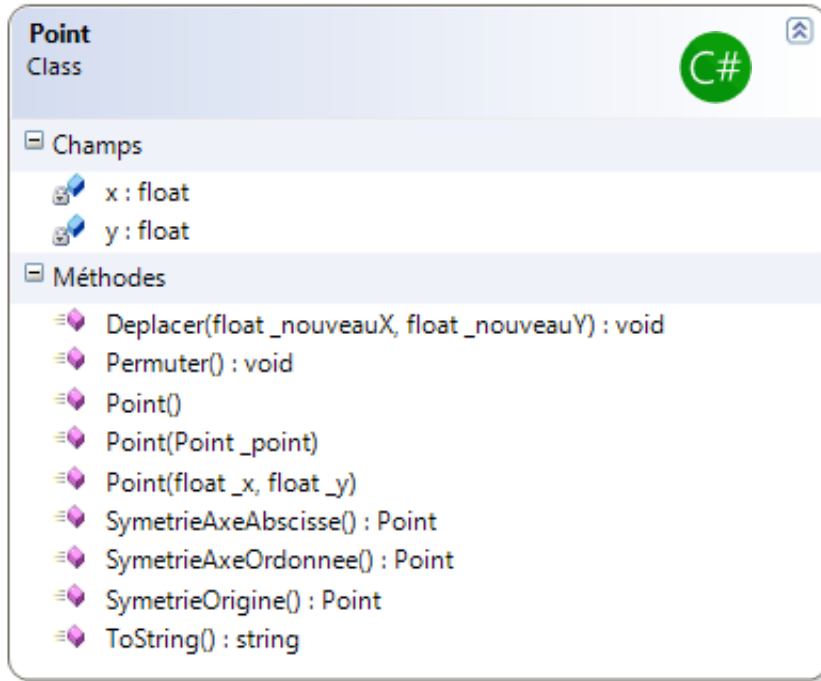
- Un point géométrique dans un espace à deux dimensions est caractérisé par son abscisse -X- et son ordonnée -Y-, valeurs réelles.
- On assigne un certain nombre de responsabilités à chaque point :
  - Se construire soit sans information ( point 0,0), ou avec une valeur pour chaque coordonnées.
  - Indiquer sa position ( abscisse et ordonnée ).
  - Se déplacer en modifiant abscisse et ordonnée.
  - Renvoyer une représentation textuelle en indiquant les valeurs de ses coordonnées.
  - Construire un point symétrique par rapport à l'axe des ordonnées.
  - Construire un point symétrique par rapport à l'axe des abscisses.
  - Construire un point symétrique par rapport à l'origine.
  - Permuter ses coordonnées ( symétrie par rapport à la bissectrice des axes Ox,Oy )

# POINT

- 1. Faire un diagramme de classe détaillé ( niveau de visibilité des membres, signature des méthodes ) sous Visual studio

# POINT

- Correction



- 2. Implémentez et tester vos constructeurs et méthodes

- Valider au près de votre formateur si il est présent!



22

# FRACTION

Notions de surcharge de constructeurs,  
d'exception

# FRACTION

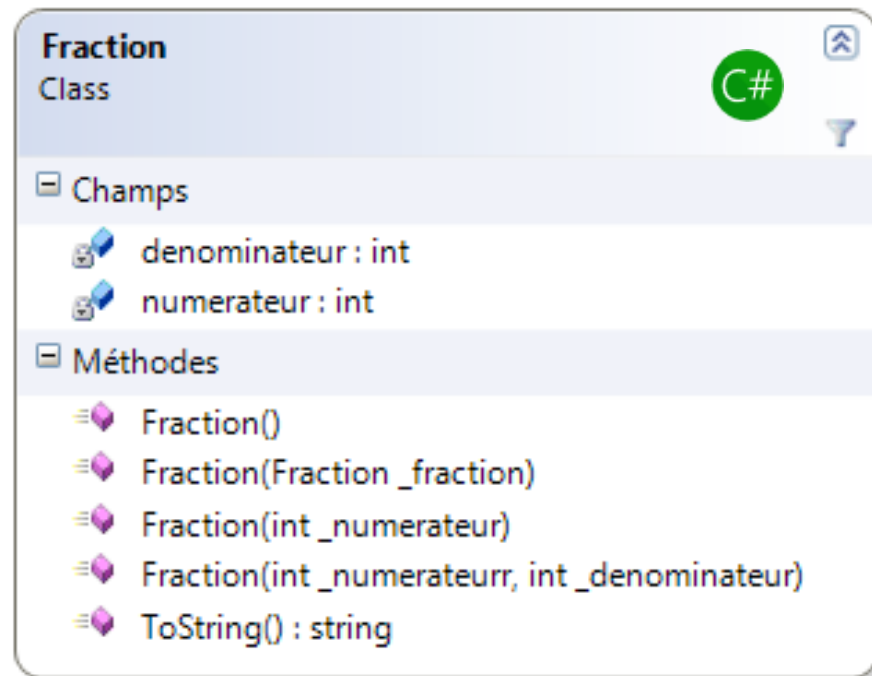
- Une fraction (ou nombre rationnel ) est un ensemble ordonné de deux entiers: numérateur et dénominateur . Nous nous proposons ici de définir une classe fraction munie de ses principales opérations.
- 1. Réalisez le diagramme de la classe Fraction. Les constructeurs suivants doivent implémenté:

```
static void Main()
{
    Fraction f1 = new Fraction(12, 7); // dans ce cas le numérateur vaudra 12 et le dénominateur 7
    Fraction f2 = new Fraction(); // fraction nulle de dénominateur 1
    Fraction f3 = new Fraction(9); // fraction dont le numérateur vaut 9 et le dénominateur 1
}
```



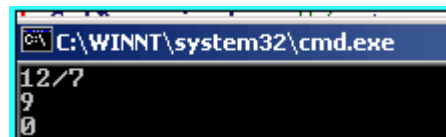
# FRACTION

## ○ Correction



# FRACTION

- 2. Ajouter dans le diagramme de classe une méthode qui retournera une représentation textuelle d'une fraction.
  - « 12/7 » pour la fraction 12/7
  - « 9 » pour la fraction 9/1
  - « 0 » pour la fraction 0/1



```
C:\WINNT\system32\cmd.exe
12/7
9
0
```



# FRACTION

- Ecrire une méthode publique **Oppose()** qui permettra d'écrire:

```
Fraction f = new Fraction(4, 7);  
f.Oppose();
```

**C#** // inverse le signe de la fraction, qui devient -4/7

- Ecrire une méthode publique **Inverse()** qui permettra d'écrire:

```
Fraction f = new Fraction(4, 7);  
f.Inverse();
```

**C#** // inverse numérateur et dénominateur, qui devient 7/4

# FRACTION

- Ecrire une méthode publique **SuperieurA** de Fraction qui permet de savoir si une :Fraction est supérieur à une autre :Fraction.

```
Fraction f = new Fraction(11, 7);  
Fraction f1 = new Fraction(5, 4);  
bool estSuperieur = f.SuperieurA(f1); // retourne vrai
```



- Ecrire une méthode publique **EgalA** de Fraction qui permet de savoir si une :Fraction est égal à une autre :Fraction.

```
Fraction f = new Fraction(11, 7);  
Fraction f1 = new Fraction(22, 14);  
bool estEgal = f.EgalA(f1); // retourne vrai
```



# FRACTION

- La méthode *privée* **Reduire()** réduit la fraction courante en divisant numérateur et dénominateur par leur pgcd et traite le problème du signe de la fraction, le signe de la fraction est le signe de son numérateur, si le numérateur et le dénominateur sont négatifs, la fraction n'a pas de signe (implicitement +).
- **On vous donne** une méthode privée **GetPgcd()** qui retourne le PGCD des numérateur et dénominateur sur la page suivante.

```
Fraction f3 = new Fraction(120, -150);  
Console.WriteLine(f3.ToDisplay()); // affiche -4/5
```

C#


# FRACTION



```
private int GetPgcd()
{
    int a = this.numerateur;
    int b = this.denominateur;
    int pgcd = 1;
    if (a != 0 && b != 0)
    {
        if (a < 0) a = -a;
        if (b < 0) b = -b;
        while (a != b)
        {
            if (a < b)
            {
                b = b-a;
            }
            else
            {
                a = a-b;
            }
        }
        pgcd = a;
    }
    return pgcd;
}
```

# FRACTION

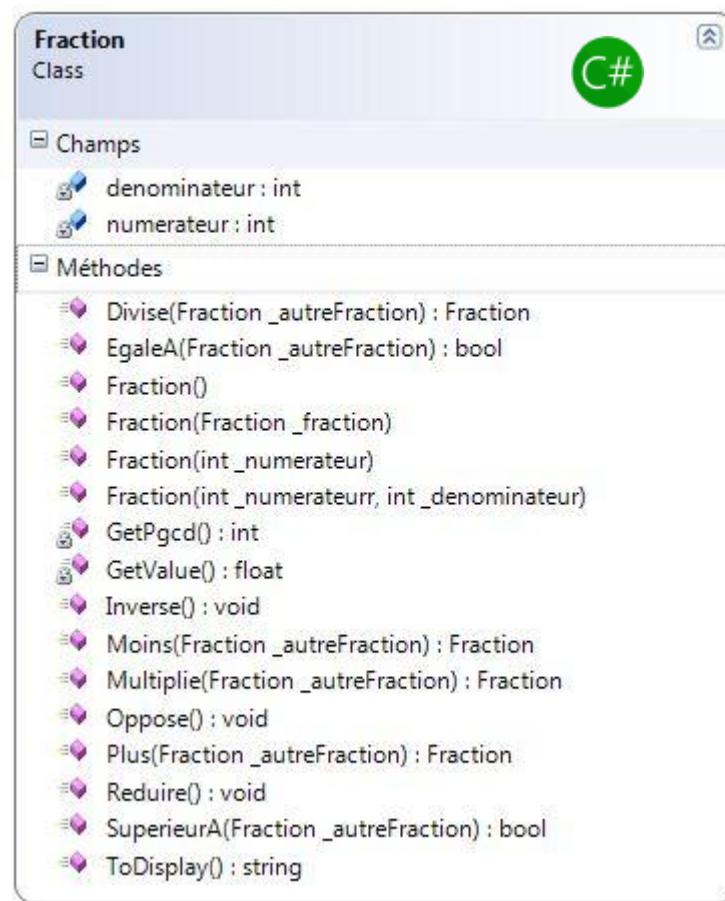
- Ecrire une méthode publique **Plus** de Fraction qui permet d'additionner une :Fraction avec une autre :Fraction.

 `Plus(Fraction _autreFraction) : Fraction`

- Ecrire une méthode publique **Moins** de Fraction qui permet de soustraire une :Fraction avec une autre :Fraction.
- Ecrire une méthode publique **Multiplie** de Fraction qui permet de multiplier une :Fraction avec une autre :Fraction.
- Ecrire une méthode publique **Divise** de Fraction qui permet de diviser une :Fraction avec une autre :Fraction. Attention, il serait préférable de réutiliser les méthodes déjà implémentées pour ce cas.
- Vérifier dans votre code que vous n'ayez pas de redondance de code!

# FRACTION

- Le diagramme a ce moment de la classe Fraction devrait être:



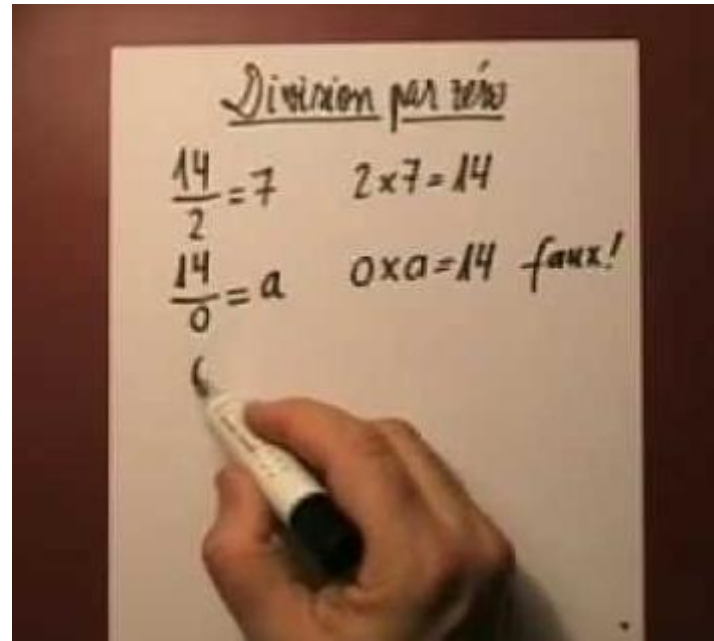
## ALLEZ PLUS LOIN (UNIQUEMENT EN C#)!

- Voici quelques idées si vous avez le temps pour aller plus loin et si vous faites du C#. La surcharge d'opérateur en Java n'existe pas!
  - Surcharge d'opérateur: permettre au codeur d'écrire  $f1 + f2$  au lieu de  $f1.Plus(f2)$

```
Fraction f = new Fraction(11, 7);  
Fraction f1 = new Fraction(22, 14);  
Fraction fplus = f1 + f2;
```



# ALLEZ PLUS LOIN!



- Chercher dans la classe Fraction les méthodes susceptibles de lever une Exception!
- Ces exceptions devront être déclarées, et générées si l'utilisateur des méthodes les maltraite!
- Indice: Il faut savoir que toute nouvelle classe d'exception hérite de la classe Exception!



- Valider au près de votre formateur si il est présent!



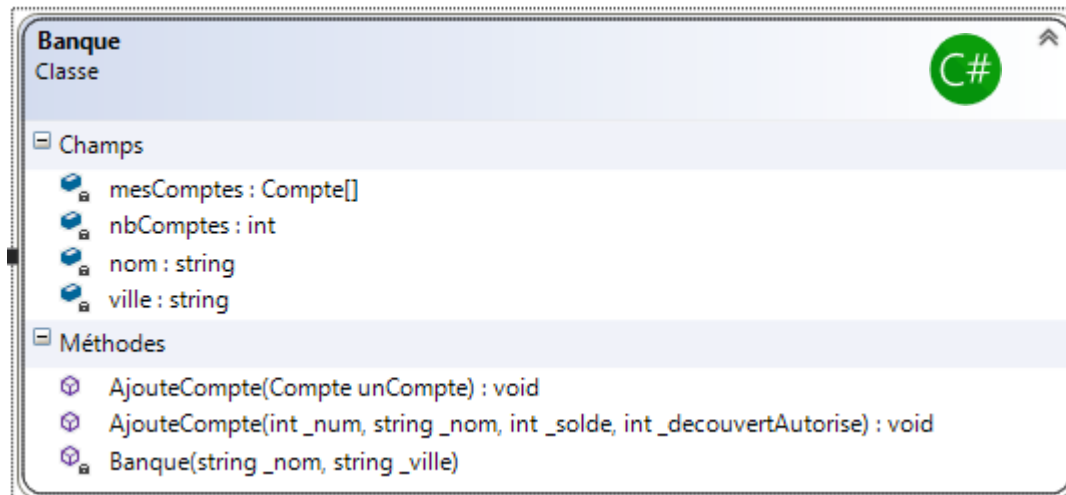
35

# BANQUE

Notion de conteneurs

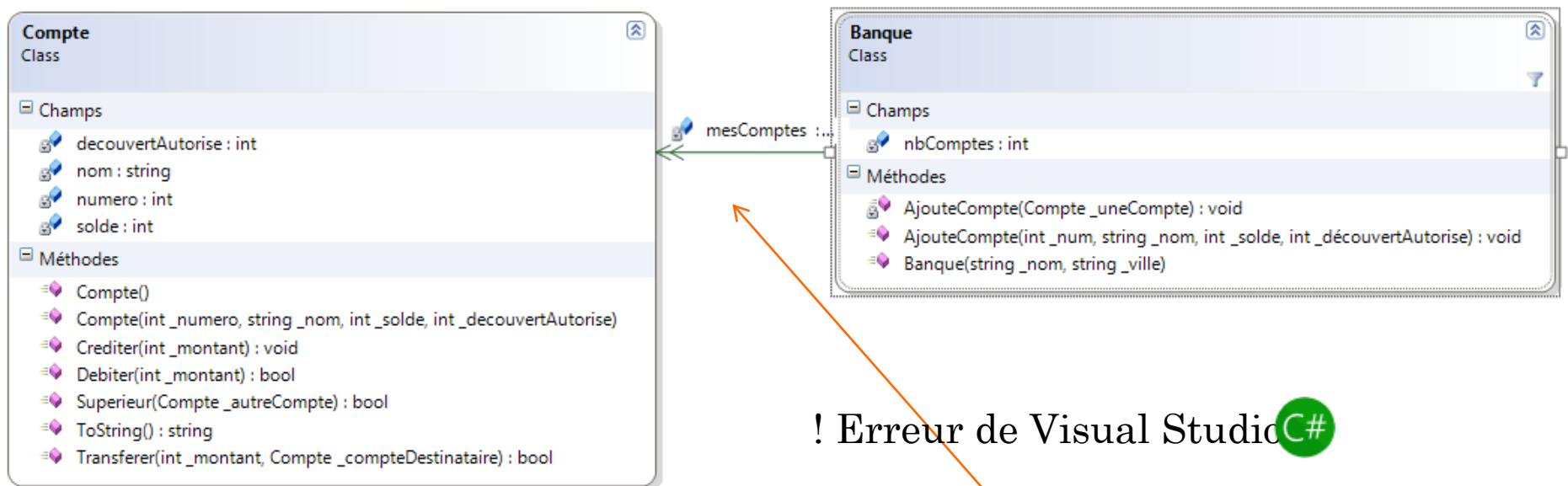
# BANQUE

- Pour cette exercice, nous allons réutiliser la classe Compte.
- Une classe Banque va permettre de regrouper les différents comptes.

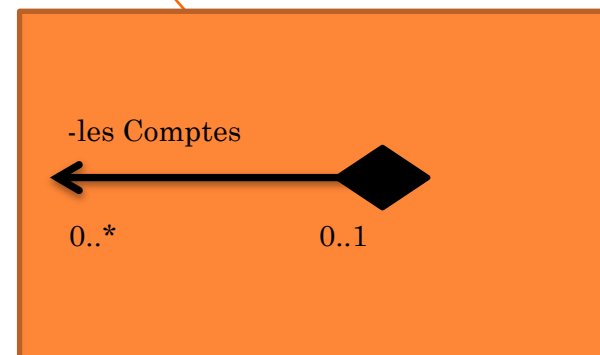


# BANQUE

## ○ Diagramme de classes



! Erreur de Visual Studio C#



# BANQUE

- Implémentez le constructeur et les deux méthodes **AjouterCompte**. Attention une des méthodes est privé!
- Ecrire la méthode **ToString** de la classe Banque qui réutilisera la méthode **ToString** de la classe Compte
- Ecrire une méthode publique **CompteSup** de la classe Banque qui retourne le compte ayant solde maximum. Vous afficherez ce compte supérieur dans la console.
- Ecrire une méthode **RendCompte** de la classe Banque qui retourne un compte en fonction de son numéro. La fonction retourne Null si le compte n'est pas trouvé. **Pour cela vous devrez ajouter éventuellement dans la classe Compte un accesseur public sur le numéro de compte.**

# BANQUE

- Ecrire une méthode qui va transférer une somme d'un compte vers un autre compte.

```
Banque b = new Banque("Credit Mututu", "Mulhouse"); C#
b.AjouteCompte(1245, "robert", 2000, 300);
b.AjouteCompte(2568, "denis", 1000, 300);
if (b.Tranferer(1245, 2568, 1000)) // 1000 euros du compte 1245 passe vers le compte 2568
{
    Console.WriteLine("transfert effectué");
}
else
{
    Console.WriteLine("transfert impossible");
}
bool resultat = b.Tranferer(1245, 2568, 5000); // retourne false ,le transfert sera impossible
```

# ALLEZ PLUS LOIN!

- Voici quelques idées si vous avez le temps pour aller plus loin
  - Utilisez un tableau dynamique typé pour mesComptes: exemple `List<Compte>` ou `ArrayList<Compte>`
  - Utilisez un tableau dynamique non typé pour mesComptes: exemple `ArrayList` ou **`java.util.Vector` pour des types non primitifs**



Pour choisir son tableau ou sa collection en Java:

<https://fmora.developpez.com/tutoriel/java/collections/introduction/>

[http://lig-membres.imag.fr/genoud/ENSJAVA/cours/supportsPDF/Collections\\_2pp.pdf](http://lig-membres.imag.fr/genoud/ENSJAVA/cours/supportsPDF/Collections_2pp.pdf)

<https://www.jmdoudoux.fr/java/dej/chap-collections.htm>

- Valider au près de votre formateur si il est présent!





## JEU 421

42

Découvertes des notions d'interfaces,  
d'encapsulation, de conteneurs et de sérialisation.

Apprendre à commenter son code!

# JEU 421

- **Règle du jeu d'un manche 421:**
- Le joueur lance 3 dés, s'il ne fait pas 421, il peut reprendre un nombre quelconque de dés et cela à deux reprises encore. Si au bout des trois lancers, il n'a pas 421, il a perdu.
- **Règle du jeu de la partie de jeu:**
- S'il gagne une manche 421, il gagne 30 points sinon il en perd 10. Au démarrage du jeu le joueur a un capital de points égal au nombre de manches choisi multiplié par 10. Une partie peut s'arrêter également si le joueur n'a plus de points.

# JEU 421

## Cas d'utilisation.

- **Pré conditions** : aucune
- **Acteur principal** : le joueur.
- **Portée** : la partie de jeux du 421
- **Niveau** : utilisateur
- **Scénario nominal** :
  - 1. Le joueur indique le nombre de manches.
  - 2. Le système initialise le jeu.
  - 3 Le système retourne la valeur visible des 3 dés triés.
  - 4. Le joueur a obtenu 421. Retour à 3.
- **Extensions.**
  - 4.1 Le joueur n'a pas obtenu 421.
    - 4.1.1 Le système demande le nombre de dés à modifier.
    - 4.1.2 Le joueur indique les dés à relancer.
    - 4.1.3 Le système relance les dés demandés. Retour à 3.
  - 4.2 C'est le troisième essai. Retour à 3.
  - 4.3 Le nombre de points disponibles passe en dessous de 0. Fin
  - 4.4 C'était la dernière manche. Fin.

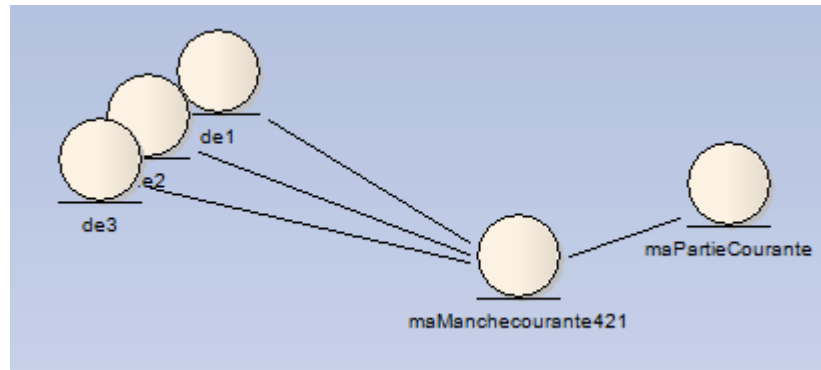
# JEU 421

- Réalisez un diagramme de collaboration entre les objets impliqués dans le jeu.
- Réalisez le diagramme de classe: vous représenterez seulement les classes sans attributs et méthodes, et leurs relations entre elles

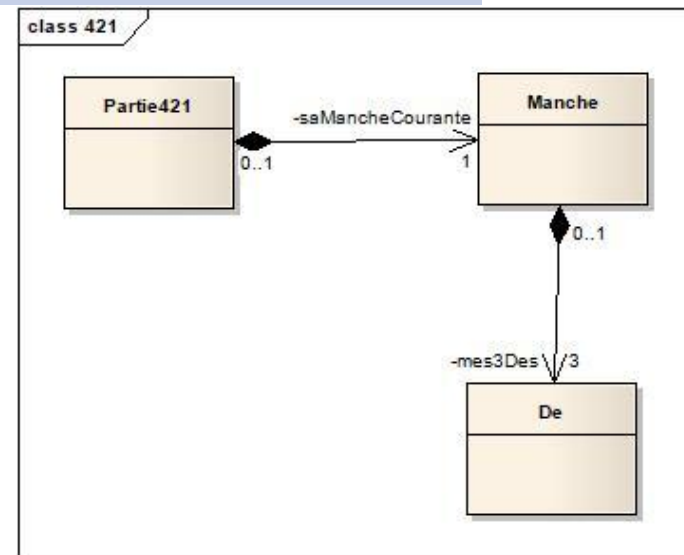
# JEU 421

## ○ Correction

- Diagramme de collaboration



- Diagramme de classes de conception



## JEU 421

- Réalisez le diagramme de classe détaillé de la classe De. Pour cela, vous devez vous poser la question:

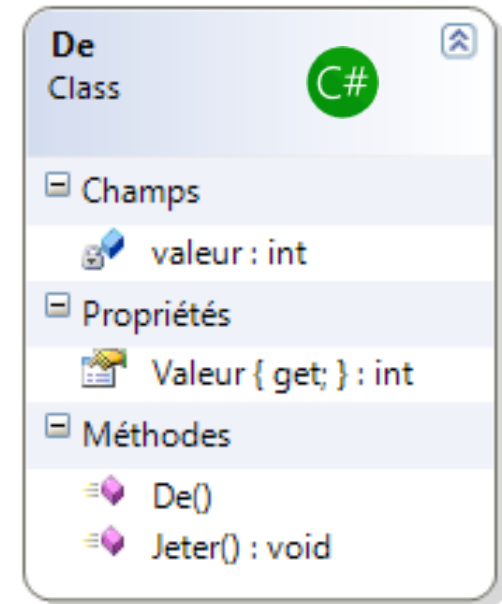
Quels peuvent être les états et les comportements d'un De?

Quels constructeurs dois-je implémenter?

Quelle est l'accessibilité des attributs et méthodes?

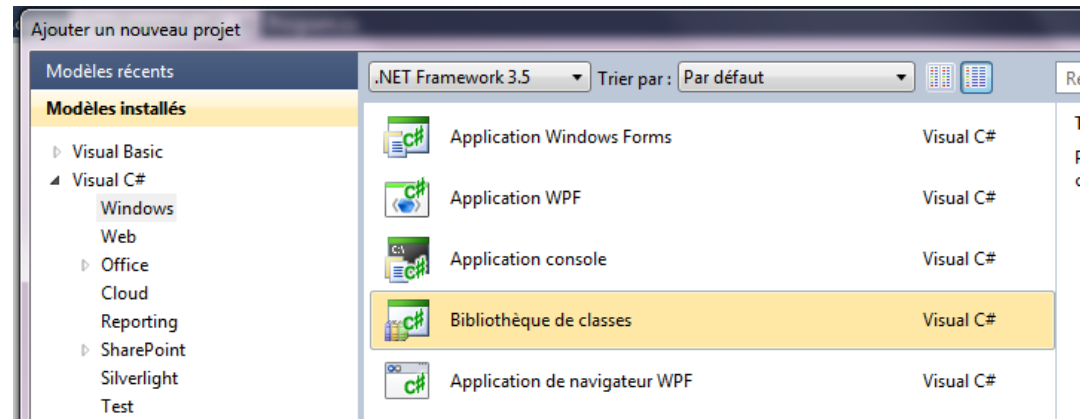
# JEU 421

- Correction:
  - On ne doit pas permettre à qq extérieur à la classe de modifier la valeur du de! Donc:
    - Un constructeur par défaut est uniquement implémenté
    - L'accesseur Get est uniquement implémenté



# JEU 421

- Vous allez implémenter la classe De. Pour cela vous devez:
  - Créer une bibliothèque de classe BibliothequeJeu421 où vous implémenterez vos classes métiers.



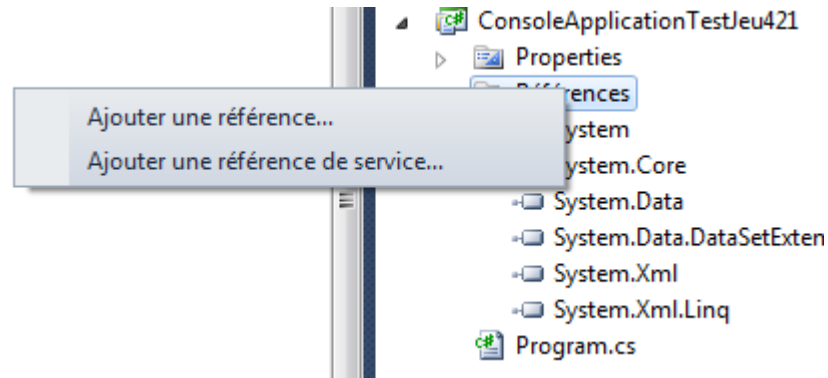
C#

- Créer une application console afin d'utiliser vos classes métiers, comme De par exemple.

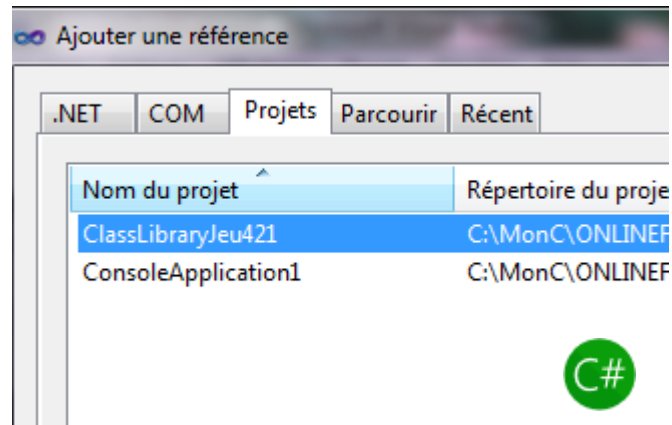


# JEU 421

- Ajouter une référence vers vos bibliothèque dans votre application console.



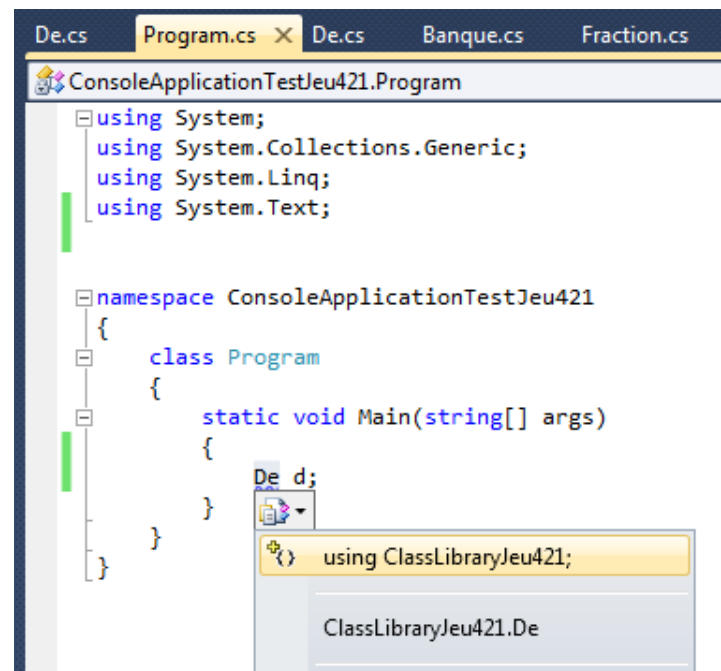
- Pour cela choisissez votre bibliothèque



# JEU 421

- N'oubliez pas de mettre une accessibilité public à votre classe De afin de pouvoir la tester et de mettre dire que vous utilisez la classe De grâce à **using**.

```
public class De  
{
```



# JEU 421

- Petite aide pour vous fournir une valeur aléatoire pour la De: la classe Alea

```
public class Alea : Random
{
    private static Alea monAlea = null;

    private Alea()
    {
    }

    public static Alea Instance()
    {
        if (monAlea == null)
        {
            monAlea = new Alea();
        }
        return monAlea;
    }

    public int Nouveau(int valMin, int valMax)
    {
        return base.Next( valMin, valMax + 1);
    }
}
```



- Utilisation de la classe Alea

```
int valeurAleatoire = Alea.Instance().Nouveau(1, 6);
```



- Question: Quelle est la conséquence sur la construction d'instance Alea de l'implémentation atypique de cette classe?
- Implémentez et tester votre classe De!

# JEU 421

- Rappel des fonctionnalités de la Manche 421:
  - Une manche 421 consiste à lancer tout d'abord 3 dés dans le but de faire la combinaison 421. Si le joueur ne réalise pas un 421 dès le premier lancé, il a la possibilité de relancer 2 fois les dés de son choix.
  - La manche 421 est stoppée dès que le joueur a atteint 3 nouveaux lancers ou dès qu'il a gagné avec la combinaison 421.
- Réalisez le diagramme de classe détaillé de la classe Manche421. Pour cela, vous devez vous poser la question:

Quels peuvent être les états et les comportements d'une Manche421?

Quels constructeurs dois-je implémenter?

Quelle est l'accessibilité des attributs et méthodes?

- Implémentez et tester votre classe Manche421!

- Valider au près de votre formateur si il est présent!

# JEU 421

- Rappel des fonctionnalités de la Partie:
  - Une Partie est initiée par un Joueur. Il fournit à la partie le nombre de Manche 421 qu'il va jouer. Pour chaque Manche 421, il recevra 10 points.
  - S'il perd la Manche 421, il perd 10 points.
  - S'il gagne la Manche 421, il gagne 30 points.
  - La manche s'arrête si il n'a plus de points ou s'il a réalisé toutes ses manches et a donc un solde positif de points.
  - On ne mémorise dans la Partie que la Manche421 courante.
- Réalisez le diagramme de classe détaillé de la classe Partie. Pour cela, vous devez vous poser la question:

Quels peuvent être les états et les comportements d'un Partie?

Quels constructeurs dois-je implémenter?

Quelle est l'accessibilité des attributs et méthodes?

- Implémentez et tester votre classe Partie!

# JEU 421

- Aide pour la lecture des choix de numéros de dés

```
string lecture = Console.ReadLine();  
string[] choix = lecture.Split(new char[] { ' ', ',', '.' });
```



```
String lecture = Scanner.nextLine();  
String[] choix = lecture.split(",")
```



- Aide pour le tri du tableau de dés:

```
Array.Sort(mes3Des);  
Array.Reverse(mes3Des);
```



```
public class De: IComparable  
{  
    public override CompareTo(Object o)  
    {  
        //a faire!  
    }  
}
```



```
Collections.sort(mes3Des);  
Collections.reverse(mes3Des);
```



```
public class De implements Comparable{  
  
    @override  
    public int compareTo(Object o){  
        //a faire!  
    }  
}
```





57

# COMMENTER SON CODE

## Dans Visual Studio



# COMMENTER LE CODE DE VOTRE BIBLIOTHÈQUE DANS VISUAL STUDIO

- Afin de documenter le code il existe un format de commentaire particulier qui indique au compilateur de traiter les commentaires comme de la documentation: il s'agit de trois slash ( / ) à la suite. Dans Visual Studio, le fait de mettre trois slash à la suite dans un endroit valide fait apparaître un menu avec les tags de documentation.



```
/// <summary>
/// Description complète de la classe. Généralement on donne la fonction de cette classe ainsi que ces particularités
/// </summary>
public class MaClasse
{
    /// <summary>
    /// Description de la variable et de son rôle dans le programme/classe.
    /// </summary>
    public static int maVariable;

    /// <summary>
    /// Description complète de la fonction. Généralement on donne le but de la fonction.
    /// </summary>
    public void UneMethode()
    {
        ...
    }

    /// <summary>
    /// Description complète de la propriété.
    /// Attention il n'est pas nécessaire de préciser si elle est en lecture seule ou non,
    /// le fait d'avoir seulement le get permet au générateur de documentation de marquer la propriété comme étant en lecture seule.
    /// </summary>
    public int UnePropriete
    {
        get
        {
            ...
        }
    }
}
```



# COMMENTER LE CODE DE VOTRE BIBLIOTHÈQUE C#

- Pour générer la documentation, il faut configurer le projet que vous voulez commenter. Allez dans le menu Propriété de votre projet.

Application

Générer\*

Événements de build

Déboguer

Ressources

Services

Paramètres

Chemins d'accès des références

Signature

Sécurité

Publier

Analyse du code

Configuration : (Debug) active Plateforme : (x86) active C#

Niveau d'avertissement : 4

Supprimer les avertissements :

Considérer les avertissements comme des erreurs

☒ Aucun

☐ Tout

☐ Avertissements spécifiques :

Sortie

Chemin de sortie : bin\Debug\ Parcourir...

☒ Fichier de documentation XML : bin\Debug\Bibliotheque421.XML

☐ Inscrire pour COM Interop

Générer un assembly de sérialisation : Auto

Options avancées...

# COMMENTER LE CODE DE VOTRE BIBLIOTHÈQUE

- Un fichier XML sera généré dans bin/debug.
- Vous avez maintenant des commentaires sur votre code

```
/// <summary>  
/// Méthode permettant de relancer des Des  
/// </summary>  
/// <param name="mesNumerosDeDes">numeros de De à relancer(1,2,3)</param>  
public void LancerDes(int[] mesNumerosDeDes)  
{
```

C#



```
Manche421 m = new Manche421();
```

m.

- Equals
- GetHashCode
- GetType
- LancerDes
- ToString

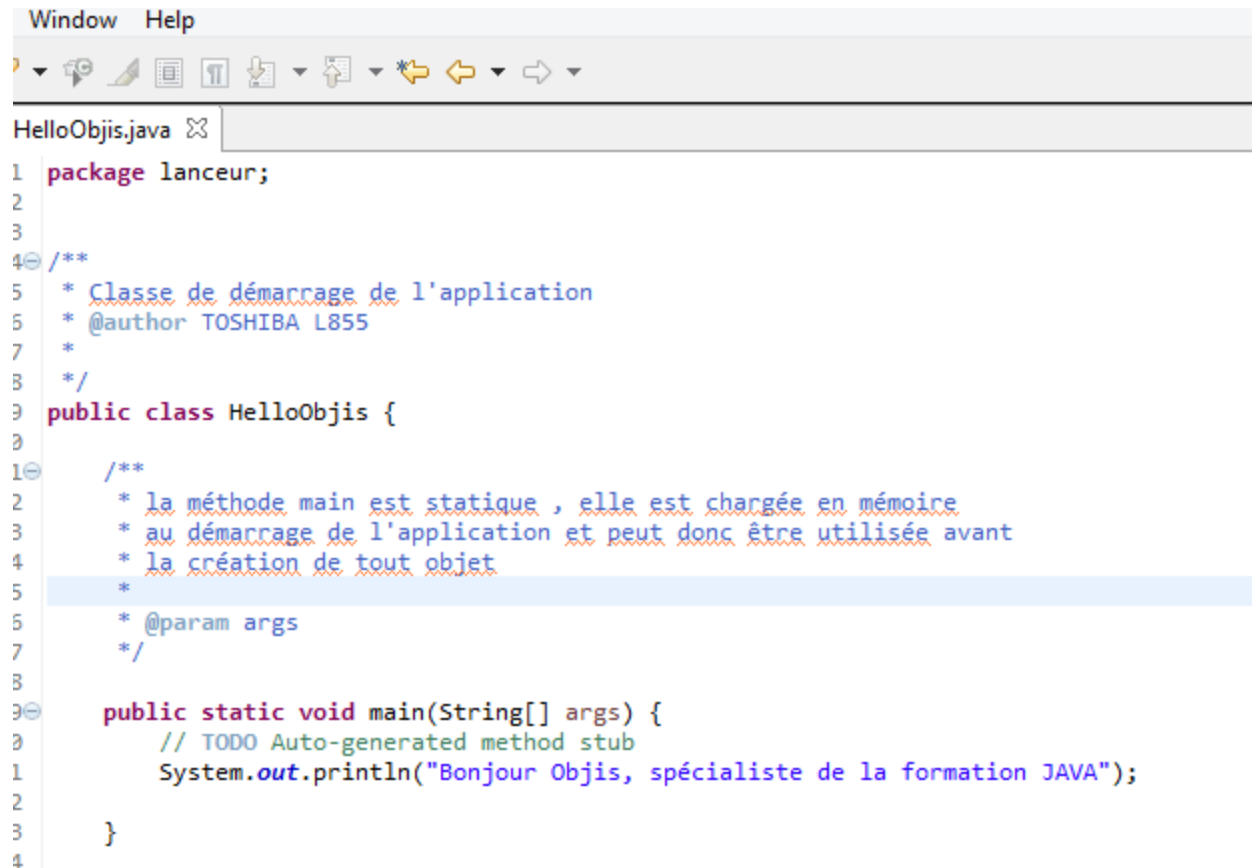
void Manche421.LancerDes(int[] mesNumerosDeDes)  
Méthode permettant de relancer des Des



61

# COMMENTER LE CODE

Dans Eclipse

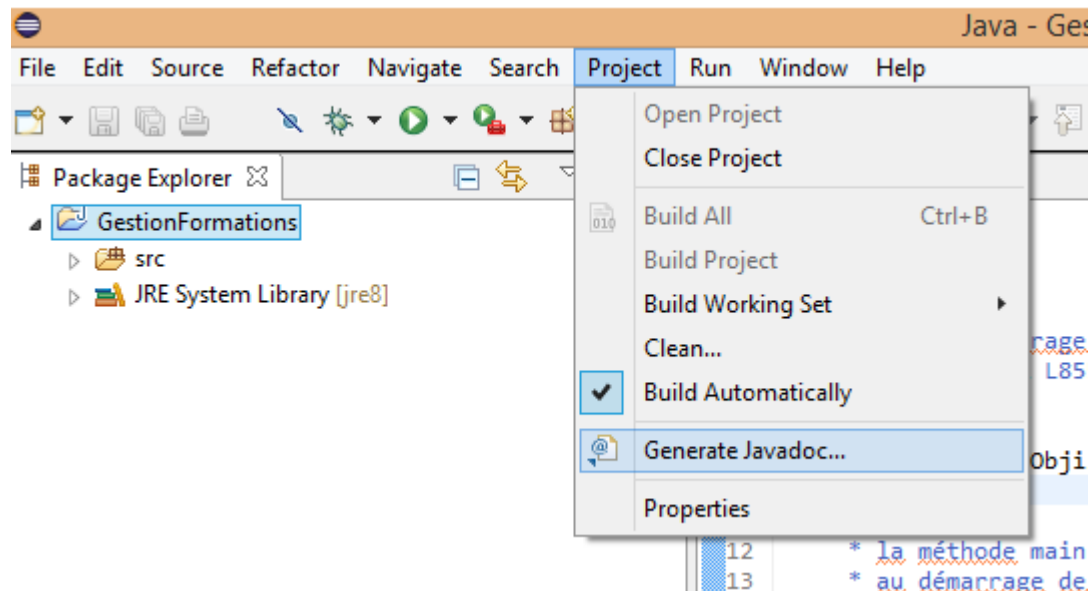


```
Window Help
HelloObjis.java
1 package lanceur;
2
3
4 /**
5  * Classe de démarrage de l'application
6  * @author TOSHIBA L855
7  *
8  */
9 public class HelloObjis {
10
11     /**
12      * la méthode main est statique , elle est chargée en mémoire
13      * au démarrage de l'application et peut donc être utilisée avant
14      * la création de tout objet
15      *
16      * @param args
17      */
18     public static void main(String[] args) {
19         // TODO Auto-generated method stub
20         System.out.println("Bonjour Objis, spécialiste de la formation JAVA");
21     }
22 }
```

- Pour commenter, on se place dans la bloc de code que l'on veut commenter et on peut utiliser le raccourci clavier suivant : **Alt+Shift+J**

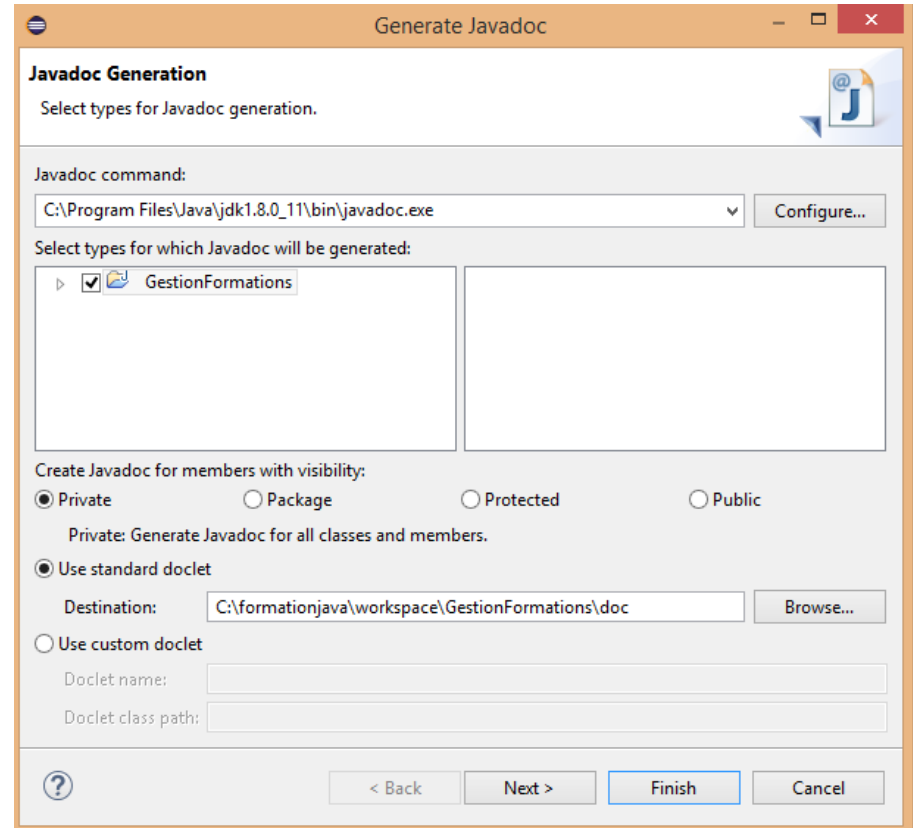
# GÉNÉRATION JAVADOC

- Sélectionnez le projet en cours et faire menu « **Project/Generate javadoc** »



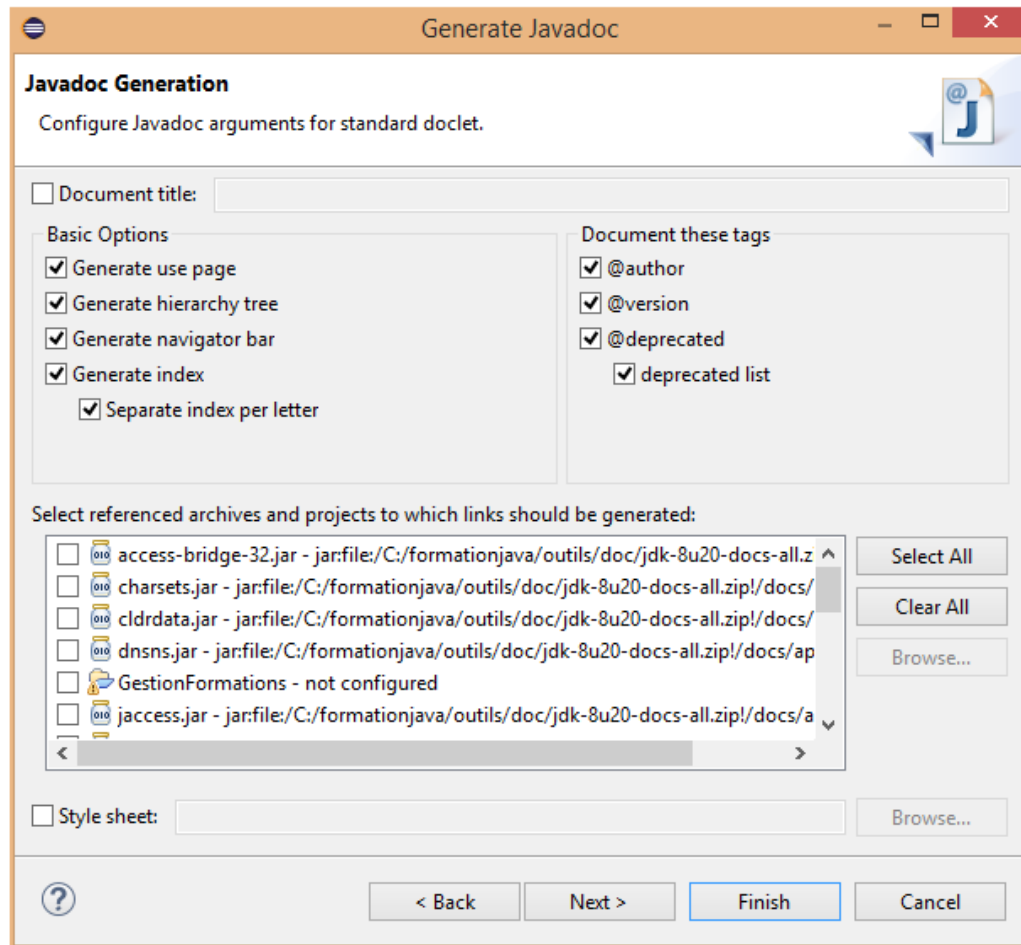
# GÉNÉRATION JAVADOC (SUITE)

- Dans l'écran qui suit la sélection « **Project/Generate javadoc** » : .
  - Dans la section 'Javadoc Command', fournissez le chemin vers l'exécutable utilisé pour la génération de la javadoc, c'est à dire de l'exécutable javadoc.exe qui se trouve dans le répertoire 'bin' du jdk . Par exemple le chemin 'C:\Program Files\Java\jdkX.X.1bin\javadoc.exe', si il n'est déjà pas présent
  - Sélectionnez la visibilité 'private' afin de créer la documentation de **toutes** les classes de l'application.
  - Vous pouvez changer éventuellement le répertoire par défaut 'doc' de destination des fichiers HTML de javadoc.



# GÉNÉRATION JAVADOC (SUITE)

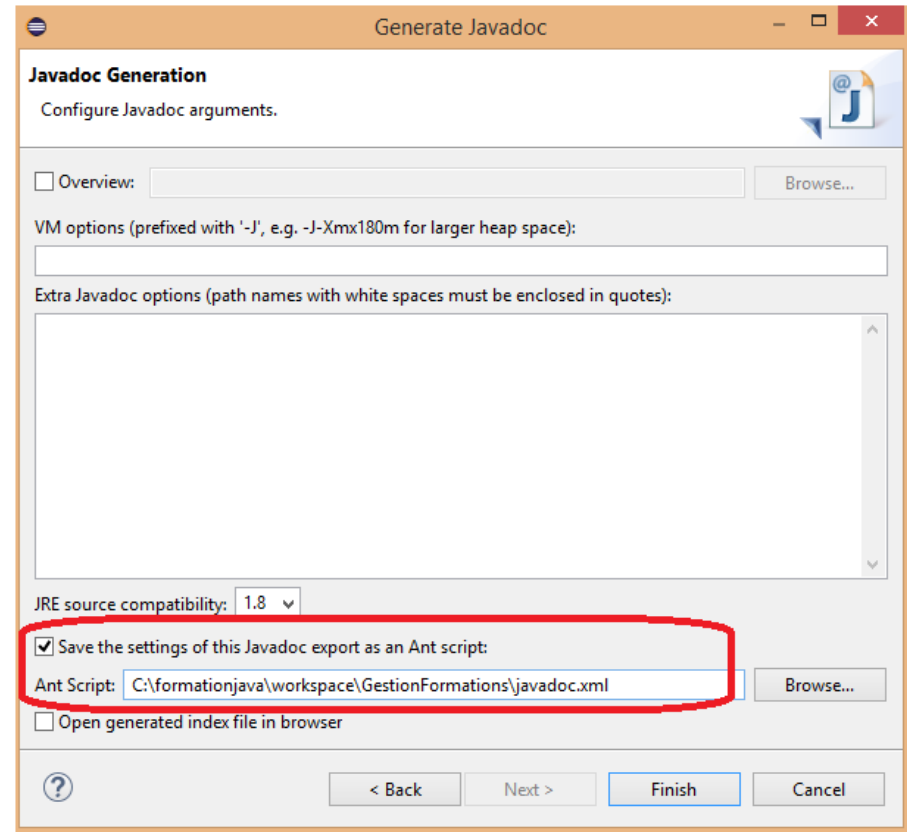
- Cliquez sur le bouton 'Next'. L'écran suivant apparaît





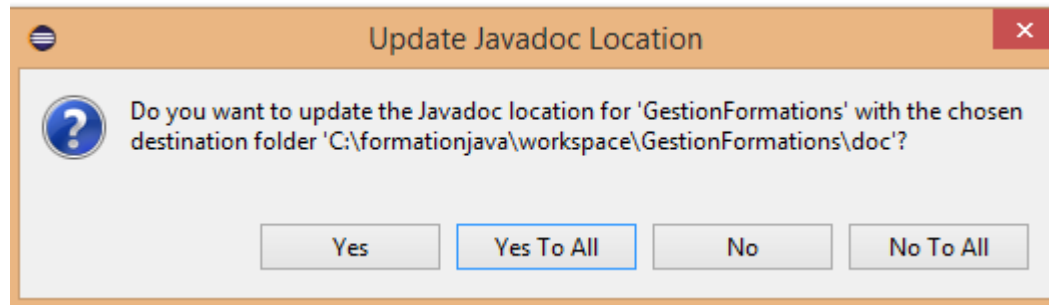
# GÉNÉRATION JAVADOC (SUITE)

- Cliquez sur le bouton 'Next'. L'écran suivant apparaît.
- Cocher l'option 'Save the settings of this javadoc export as an Ant script'. Cela permet d'enregistrer la configuration de génération javadoc dans un script ant.
- Pour régénérer la javadoc ultérieurement, il suffira d'exécuter le script (click droit sur le script javadoc.xml / Run / Ant Build)
- Cliquez sur le bouton 'Finish'.

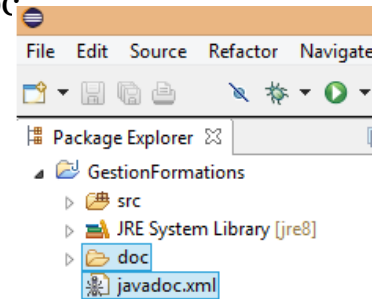


# GÉNÉRATION JAVADOC (SUITE)

- L'écran suivant apparaît.



- Validez (Bouton YES) ce message qui informe de la création du répertoire doc. Le message suivant apparaît, informant de la création du fichier ANT javadoc.xml, clé de l'automatisation des futures javadoc.



- Pour rafraichir cette documentation, Click droit / Refresh sur le répertoire 'doc'

# GÉNÉRATION JAVADOC (SUITE)

- Cliquer droit sur fichier 'index.html', et faire Open with, Web Browser pour voir la documentation!

The screenshot illustrates the Javadoc generation process in an IDE. On the left, the Package Explorer shows the project structure, with the 'doc' directory expanded and 'index.html' selected. A large orange arrow points from 'index.html' to the Javadoc viewer on the right. The Javadoc viewer displays the documentation for the 'HelloObjis' class, including the class signature, inheritance, and a constructor summary.

**Package Explorer:**

- GestionFormations
  - src
  - JRE System Library [jre8]
  - doc
    - index-files
    - lanceur
      - allclasses-frame.html
      - allclasses-noframe.html
      - constant-values.html
      - deprecated-list.html
      - help-doc.html
      - index.html**
      - overview-tree.html
      - package-list
      - script.js
      - stylesheet.css
  - javadoc.xml

**Javadoc Viewer:**

file:///C:/formationjava/workspace/GestionFormations/doc/index

**All Classes**

HelloObjis

**PACKAGE CLASS USE TREE DEPRECATED INDEX HEL**

**PREV CLASS NEXT CLASS FRAMES NO FRAMES**

**SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL:**

**lanceur**

**Class HelloObjis**

**java.lang.Object**

**lanceur.HelloObjis**

**public class HelloObjis**

**extends java.lang.Object**

Classe de démarrage de l'application

**Author:**

TOSHIBA L855

**Constructor Summary**

**Constructors**

**Constructor and Description**

**HelloObjis ()**

**Method Summary**

- Valider au près de votre formateur si il est présent!

# ALLEZ PLUS LOIN!

- Voici quelques idées si vous avez le temps pour aller plus loin
  - Ajouter une classe Joueur dans l'objectif de mémoriser le score des joueurs.
  - Vous sérialiserez les joueurs dans un fichier .bin pour .xml. La sérialisation consiste à écrire l'état d'un objet dans un fichier et la dé sérialisation consiste à reconstruire l'objet à partir du fichier.
  - Ce fichier devra se trouver dans le dossier application data de l'ordinateur.

```
string path = System.Environment.GetFolderPath(System.Environment.SpecialFolder.ApplicationData);
```



```
String dataFolder = System.getenv("APPDATA");
```



- Pour une sérialisation binaire:
  - En C#, vous utiliserez la Classe **BinaryFormatter** pour sérialiser le tableau de Joueur en binaire. Ce travail nécessite un peu de recherche sur internet.
  - En Java, Pour pouvoir être sérialisée, vous utiliserez la Classe **ObjectOutputStream**, et les classes serialisables devront implémenter l'interface **java.io.Serializable**. Ce travail nécessite un peu de recherche sur internet.
- Recommencez pour une sérialisation XML!

- Valider au près de votre formateur si il est présent!

# LES AUTRES COMPTES

Notions d'héritage

# LES AUTRES COMPTES

- A la banque, il y a des comptes classiques, des comptes permettant les découverts, des comptes rémunérés, etc..
- Vous avez actuellement codé un compte permettant les découverts mais ce n'est pas toujours le cas.
- Proposez une réorganisation de votre conception UML de la vision des comptes permettant de différencier ces trois types de comptes. Vous pouvez avoir une définition d'un compte rémunéré sur le prochain slide.



# LES AUTRES COMPTES-COMPTÉ RÉMUNÉRÉE

- Un compte rémunéré possède un numéro, un nom d'utilisateur, un solde, un découvert autorisé. Tous les comptes rémunérés sont soumis au même taux de rémunération; par ailleurs pour calculer à tout moment les intérêts produits, il est nécessaire de connaître la date d'ouverture du compte rémunéré.
- Lorsque le compte rémunéré est interrogé sur son solde, la valeur retournée est le solde plus les intérêts.
- On peut débiter ou créditer ces comptes ainsi que transférer d'un compte à un autre; un compte peut comparer son solde avec le solde d'un autre compte.
- Chaque compte peut afficher ses informations. Enfin chaque compte rémunéré peut retourner les intérêts produits.

# LES COMPTES ET LES AUTRES OFFRES

- Tous les comptes doivent être capable de générer un nom de compte à partir de son numéro, de son type et des 3 premières lettres du nom du propriétaire.
  - CD 134672144687 DUP
  - CC 846134621666 MAR
  - CR 256798314779 THY

*Pour réaliser ce comportement, est il préférable de choisir une méthode abstraite ou virtuelle?*

- Tous comptes, produits bancaires (livrets, assurances vies, produits boursiers), crédits ont des frais de clôture. Les règles de calculs de ces frais sont propres à chacun, et ces différentes offres fournissent aux clients n'ont rien de commun, à part qu'ils ont un propriétaire.

*Comment peut-on prendre en compte dans notre modèle le calcul des frais de clôture de compte?*

# BIBLIOTHEQUE

- <https://actufinance.fr/guide-banque/sommaire-activites-bancaires.html>