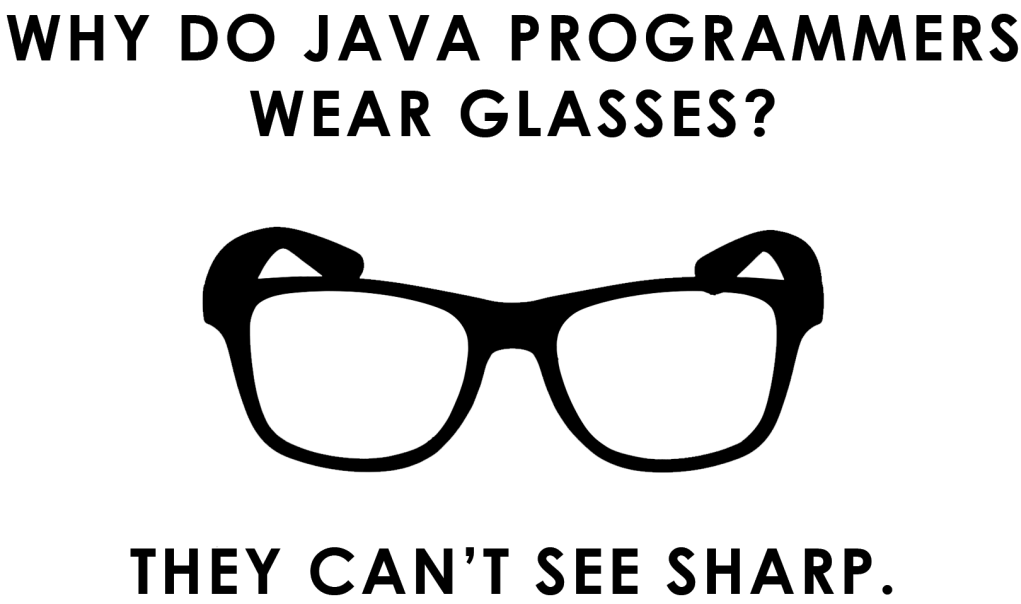
How to beat an ACDC 101

Florian Amsallem & Tom Decrette

Aujourd'hui, nous allons voir les bases du C# et de la POO. L'objectif est de se familiariser avec la syntaxe du C# et de comprendre le fonctionnement de la programmation orientée objet (POO). Pour cela, vous allez devoir dans un premier temps implémenter des fonctions classiques puis créer des classes et manipuler celles-ci.



1 Partie 1

Voici un ensemble d'exercices utilisant les principes de base du C#. Ils ne sont **pas** triés par ordre de difficulté. Si vous êtes bloqués, demandez de l'aide aux assistants.

1.1 Fibonacci

Le but de ce premier exercice est d'implémenter la suite de Fibonacci en C#. Pour rappel, la suite de Fibonacci est définie comme suit :

$$\mathscr{F}_0 = 0$$

$$\mathscr{F}_1 = 1$$

$$\mathscr{F}_{n+2} = \mathscr{F}_n + \mathscr{F}_{n+1}$$

```
1  private static long Fibo(long n)
2  {
3      // TODO
4      return 0;
5  }
```

1.2 Factorielle

Pour cet exercice, vous devez implémenter la fonction factorielle.
Pour rappel, factorielle est décrite comme suit :

$$\mathscr{T}(0) = 1$$

$$\mathscr{T}(n) = n \times \mathscr{T}(n-1)$$

```
1  private static long Fact(long n)
2  {
3      // TODO
4      return 0;
5  }
```

1.3 Swapons

Pour cet exercice, vous allez devoir utiliser les références. Le but est d'échanger les valeurs de x et de y.

```
1  private static void Swap(ref int x, ref int y)
2  {
3      // TODO
4  }
```

1.4 MinTab

Nous allons utiliser les tableaux. Votre but est de trouver la plus petite valeur dans un tableau de **long**.

```
1  private static long MinTab(long[] tab)
2  {
3      // TODO
4      return 0;
5  }
```

1.5 Sum

Il est possible en C# de déclarer des matrices. Par exemple, pour déclarer la matrice identité de taille 3 :

```
1  int[,] identity3 = new int[,]
2  {
3      {1, 0, 0},
4      {0, 1, 0},
5      {0, 0, 1}
6  };
```

Pour cet exercice, vous devez calculer la somme de deux matrices de même dimension.
Rappel :

$$\mathscr{A}+\mathscr{B} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \dots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \dots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \dots & a_{mn}+b_{mn} \end{pmatrix}$$

```
1  private static long[,] Sum(long[,] mat1, long[,] mat2)
2  {
3      long[,] matRes;
4      // TODO
5      return matRes;
6  }
```

1.6 Parité

Pour cet exercice, nous allons utiliser les **listes**. Vous devez vérifier qu'une liste donnée respecte la parité ou non.

On dit qu'une liste respecte la parité si elle est composée d'une alternance d'entiers pairs et impairs en commençant par un entier pair.
Par exemple :

```
1  [4, 5, 8, -3, 2, 5] // Est une liste qui respecte la parité.
2  [4, 5, 8, -3, 1, 5] // N'est pas une liste qui respecte la parité.
3  [5, 8, -3, 2, 5, 4] // N'est pas une liste qui respecte la parité.
```

Voici le prototype de la fonction :

```
1  private static bool Parite(List<int> list)
2  {
3      // TODO
4      return true;
5  }
```

1.7 Bits Counter

Vous devez compter le nombre de bits mis à 1 dans un nombre donné.

Conseil : Utilisez les opérateurs bitwise.

```
1  private static uint NbBitsSet(long number)
2  {
3      // TODO
4      return 0;
5  }
```

2 Partie 2

Les choses sérieuses vont commencer! Vous allez devoir créer vos premières classes et objets.
Le but de cet exercice est d'implémenter les classes :
— Student
— ACDC
— Sup
— Fight
— Arena

Une fois que cela est fait, nous allons pouvoir faire combattre nos étudiants les uns contre les autres!

2.1 Student

Comme vous pouvez l'imaginer la classe `Student` va représenter un étudiant. Celui-ci possède des points de vie, des dégâts, etc...

2.1.1 Student Constructor

Vous l'avez compris vous allez implémenter votre premier constructeur. Mais avant cela, il faut définir les attributs de la classe.

Voici la liste des attributs et leur type :
— name : `string`
— life : `int`
— damage : `int`
— isMagician : `bool`
— physicalArmor : `int`
— magicalArmor : `int`

Vous pouvez mettre la protection des attributs à **public** pour le moment.

```
1  class Student
2  {
3      // Attributs (A vous de compléter)
4      public string name;
5      // ...
6
7      // Constructeur (A vous de compléter)
8      public Student(string name, /* ... */)
9      {
10         this.name = name;
11         // ...
12     }
13 }
```

2.2 Take damage

Maintenant que nos étudiants ont des attributs et peuvent être instanciés (construits), vous allez implémenter une méthode.

La méthode `TakeDamage` va appliquer des dégâts (magiques ou physiques) à notre étudiant.
Voici la formule à appliquer :

$$life = life - (damage - armor)$$

Il faut compléter et modifier la formule de manière à respecter les points suivants :
— Armor dépend du type de dégât.
— L'étudiant ne peut pas gagner de la vie.
— L'étudiant ne peut pas avoir une vie négative.

```
1  public void TakeDamage(int damage, bool isMagical)
2  {
3      // TODO
4  }
```

2.3 Attack

Nous souhaitons maintenant pouvoir attaquer d'autre étudiants.
Implémentez la fonction `Attack` qui prend un étudiant en paramètre et l'attaque.

⚠**Attention** un étudiant ne peut pas s'attaquer lui-même! Lancer une exception dans ce cas.

```
1  public void Attack(Student s)
2  {
3      // TODO
4  }
```

2.4 Status

Pour connaître l'état d'un étudiant, nous allons faire une procédure qui affiche dans la console l'état de l'étudiant. Par exemple, *"I still have 42 HP."*.

```
1  public void Status()
2  {
3      // TODO
4  }
```

2.5 Counter

Nous souhaitons contrôler le nombre d'élèves via une variable `nbStudent`.
Déclarez et incrémentez cette variable de manière à ce qu'elle compte le nombre d'élèves.
Vous pouvez faire la procédure `DisplayNbStudent` qui affiche le nombre d'étudiants. Par exemple, *"There are 42 student(s)."*
Tips : Utilisez le mot clé `static`.

2.6 Getters & Setters

Pour cette classe nous souhaitons pouvoir contrôler les valeurs de certains attributs. De ce fait, les attributs suivants devront être passés en *private* : **name, life, dammage.**
Ainsi, comme vous avez pu le voir en conférence, vous allez devoir créer vos propres getters & setters.
Pour la vie, nous souhaitons contrôler que dans aucun cas, l'objet puisse avoir une vie négative. Pour cela, voici un exemple :

```
public string Life
{
    get { return life_; }
    set { life_ = Math.Max(0, value); }
}
```

Maintenant, à vous de faire de même pour que, quelque soit le nom entré par l'utilisateur, l'attribut `name` de l'objet soit toujours en majuscule.

```
public string Name
{
    // TODO
}
```

Quant aux dégâts, ceux-ci ne pourront jamais excéder 10 ou être inférieur à 0.

```
public int Damage
{
    // TODO
}
```

2.7 Tests

Pour pouvoir tester vos fonctions, vous pouvez utiliser les appels des fonctions suivantes dans votre main :

```
public static void Main(string[] args)
{
    Student flomonster = new Student("Flomonster", 100, 10, true, 0, 5);
    Student theo = new Student("Theo", 75, 7, false, 5, 0);
    Student.DisplayNbStudent();
    flomonster.Name = "Florian";
    flomonster.Attack(theo);
    theo.Status();
    flomonster.Damage = 30;
    try
    {
        theo.Attack(theo);
    }
    catch (Exception e)
    {
        Console.WriteLine("You tried to attack yourself.");
    }
    while (theo.Life != 0)
    {
        flomonster.Attack(theo);
        theo.Status();
    }
}
```

Et cela doit afficher le résultat suivant :

```
There are 2 student(s).
THEO: I still have 65 HP.
You tried to attack yourself.
THEO: I still have 55 HP.
THEO: I still have 45 HP.
THEO: I still have 35 HP.
THEO: I still have 25 HP.
THEO: I still have 15 HP.
THEO: I still have 5 HP.
THEO: I still have 0 HP.
```

2.8 Classe Sup

Nous allons pouvoir passer à notre deuxième classe, la classe `Sup`. Maintenant que vous êtes des experts, vous allez implémenter les attributs, constructeurs et méthodes sans squelette :

Voici la description de la classe `Sup` :
— Un Sup est un étudiant (la classe hérite donc de la classe `Student`).
— Un Sup n'est pas magicien.
— La variable `nbSup` compte le nombre de Sup instanciés.
— La procédure `DisplayNbSup` affiche le nombre de Sup instanciés. Par exemple, *"There are 42 Sups."*
— La procédure `Status` n'a pas le même comportement que la classe `Student`. Par exemple, elle affiche *"ALEXANDRE : Please help! I have 42 HP left."*.

Rappel : N'hésitez pas à appeler un assistant.

2.8.1 Tests

```
1  public static void Main(string[] args)
2  {
3      List<Sup> listSup = new List<Sup>();
4      for (int i = 0; i < 10; i++)
5      {
6          listSup.Add(new Sup("Alexandre", 75, 7, 0, 5));
7          Sup.DisplayNbSup();
8      }
9      int damage = 5;
10     foreach (Sup sup in listSup)
11     {
12         sup.TakeDamage(damage, true);
13         damage += 5;
14         sup.Status();
15     }
16 }
```

Cela est censé produire le résultat suivant :

```
There are 1 Sups.
There are 2 Sups.
There are 3 Sups.
There are 4 Sups.
There are 5 Sups.
There are 6 Sups.
There are 7 Sups.
There are 8 Sups.
There are 9 Sups.
There are 10 Sups.
ALEXANDRE: Please help! I have 75 HP left.
ALEXANDRE: Please help! I have 70 HP left.
ALEXANDRE: Please help! I have 65 HP left.
ALEXANDRE: Please help! I have 60 HP left.
ALEXANDRE: Please help! I have 55 HP left.
ALEXANDRE: Please help! I have 50 HP left.
ALEXANDRE: Please help! I have 45 HP left.
ALEXANDRE: Please help! I have 40 HP left.
ALEXANDRE: Please help! I have 35 HP left.
ALEXANDRE: Please help! I have 30 HP left.
```

2.9 Classe ACDC

La Classe `ACDC` va elle aussi dépendre de la classe `Student`, cependant elle n'est pas encore présente dans le squelette : il va donc falloir la créer. La description de la classe est la suivante :

- Un ACDC est un étudiant (la classe hérite donc de la classe `Student`).
- Un ACDC est un magicien.
- Tous les autres attributs restent inchangés.
- La variable `nbACDC` compte le nombre d'ACDC instanciés.
- La procédure `DisplayACDC` affiche le nombre d'ACDC instanciés. Par exemple, *"There are 24 ACDC."*
- La procédure `Status` n'a pas le même comportement que la classe `Student`. Par exemple, elle affiche *"FLORIAN : You can't beat me, I have 42 HP left."*.

2.9.1 Tests

Voici une suite de tests :

```
1  public static void Main(string[] args)
2  {
3      List<ACDC> listACDC = new List<ACDC>();
4      for (int i = 0; i < 5; i++)
5      {
6          listACDC.Add(new ACDC("Florian", 100, 10, 5, 5));
7          ACDC.DisplayNbACDC();
8      }
9      int damage = 5;
10     foreach (ACDC acdc in listACDC)
11     {
12         acdc.TakeDamage(damage, false);
13         damage += 10;
14         acdc.Status();
15     }
16 }
```

Cela est censé produire le résultat suivant :

```
There are 1 ACDC.
There are 2 ACDC.
There are 3 ACDC.
There are 4 ACDC.
There are 5 ACDC.
FLORIAN: You can't beat me, I still have 100 HP.
FLORIAN: You can't beat me, I still have 90 HP.
FLORIAN: You can't beat me, I still have 80 HP.
FLORIAN: You can't beat me, I still have 70 HP.
FLORIAN: You can't beat me, I still have 60 HP.
```

2.10 Classe Fight

Laissez place au combat! On continue avec la classe `Fight` qui va permettre de gérer les combats entre étudiants.

— Une `Fight` possède deux étudiants (`student1` et `student2`).
— Une `Fight` possède un compteur de tours `round` de type `uint`.

⚠Tous ces attributs ne doivent pas être modifiables. Ils peuvent cependant être accessibles.

2.10.1 Is it finished?

La méthode `isFinished` renvoie `true` si l'un des étudiants a une vie nulle; autrement, elle renvoie `false`.

```
1  public bool isFinished()
2  {
3      // TODO
4      return false;
5  }
```

2.10.2 Update

La méthode `Update` simule l'attaque d'un étudiant sur l'autre puis l'inverse. Si `verbose` est à `true`, alors la méthode affiche le `round` courant et l'état des étudiants. Par exemple :

```
- Round 42
FLOMONSTER: You can't beat me, I still have 100 HP.
THIBAULT: Please help! I have 5 HP left.
```

```
1  public void Update(bool verbose)
2  {
3      // TODO
4  }
```

2.10.3 GetWinner

Cette méthode renvoie l'étudiant vainqueur du combat. S'il n'y a pas encore de gagnant ou si les deux étudiants sont morts, la fonction renvoie `null`.

```
1  public Student GetWinner()
2  {
3      // TODO
4      return null;
5  }
```

2.10.4 Finish

Cette méthode écrit dans la console le résultat du combat. Par exemple :

```
// Si le combat est fini
The fight is done.
FLOMONSTER won this fight!
// Sinon
No one won this fight!
```

2.10.5 Tests

Voici de quoi tester votre code :

```
1  Student flomonster = new Student("Flomonster", 100, 10, true, 0, 5);
2  Student theo = new Student("Theo", 75, 7, false, 5, 0);
3  Fight fight = new Fight(flomonster, theo);
4  while (!fight.isFinished())
5      fight.Update(true);
6  Console.WriteLine("Winner: {0}", fight.GetWinner().Name);
7  fight.Finish();
```

```
-- Round 1 --
FLOMONSTER: I still have 93 HP.
THEO: I still have 65 HP.
-- Round 2 --
FLOMONSTER: I still have 86 HP.
THEO: I still have 55 HP.
-- Round 3 --
FLOMONSTER: I still have 79 HP.
THEO: I still have 45 HP.
-- Round 4 --
FLOMONSTER: I still have 72 HP.
THEO: I still have 35 HP.
-- Round 5 --
FLOMONSTER: I still have 65 HP.
THEO: I still have 25 HP.
-- Round 6 --
FLOMONSTER: I still have 58 HP.
THEO: I still have 15 HP.
-- Round 7 --
FLOMONSTER: I still have 51 HP.
THEO: I still have 5 HP.
-- Round 8 --
FLOMONSTER: I still have 44 HP.
THEO: I still have 0 HP.
Winner: FLOMONSTER
-- The fight is done --
FLOMONSTER won this fight!
```

2.11 Classe Arena

Maintenant nous allons pouvoir créer une classe `Arena` qui réalisera un certain nombre de matchs. Pour stocker l'ensemble des matchs, nous allons utiliser une Pile. Cet attribut s'appelle `matchUp` il est donc de type `Stack<Fight>`

Hint

> Pour plus d'informations sur l'utilisation des piles en C# , vous pouvez aller voir le fonctionnement de la classe `Stack` sur le site MSDN.

2.11.1 Constructeurs

Pour cette classe, nous allons utiliser deux différents constructeurs.
Le premier constructeur prend en paramètre directement la pile de combats.

```
1  public Arena(Stack<Fight> matchup)
2  {
3      // TODO
4  }
```

Pour ce qui est du second constructeur, il prend en paramètre un `uint` qui représente alors le nombre de combats à créer. Pour créer un combat vous devez :

— Créer un Sup avec des attributs aléatoire.
— Créer un ACDC avec des attributs aléatoire.
— Créer le "fight" entre le Sup et l'ACDC.
— Empiler le nouveau "fight" dans la pile.

Hint

Allez voir du côté de la classe `Random` sur MSDN.

```
public Arena(uint nbFight)
{
    // TODO
}
```

2.11.2 Méthode

La classe `Arena` ne comporte qu'une méthode qui va lancer tous les combats à la suite et indiquer le nombre de victoire pour les ACDC, le nombre de victoire pour les Sups, et le nombre d'égalités.

```
public void ResolveFights()
{
    // TODO
}
```

2.11.3 Tests

Voici de quoi tester votre code :

```
Arena arena = new Arena(100);
arena.ResolveFights();
```

```
ACDC: 100 wins / Sups: 0 wins / Draws: 0
```

2.12 Bonus

Vous restez libre d'implémenter en plus toutes les fonctions/classes que vous voulez. Montrez-nous vos compétences, épatez nous, ça ne sera que bénéfique pour vous.

These violent deadlines have violent ends.