Mise en bouche

Hello World !

Pour commencer, nous allons Re-coder le classique Hello World, implémentez la fonction éponyme qui affiche 'Hello World !' sur la sortie standart.

```
static void HelloWorld();
```

Notez qu'il ne faut pas recopier le ';'. Il est ici utilisé car nous déclarons le **prototype** de la fonction (son nom, ses arguments, sa visibilité, sa valeur de retour…).

N'hésitez pas à vous renseignez sur la méthode Console.WriteLine

Programmation Impérative

Print Array

Fonction assez pratique puisqu'elle permet de facilement inspecter les éléments d'un tableau (jusqu'a la découvert d'un débugger).

Prototype

```
static void PrintArray(int[] array);
```

Example

```
// int[] test = new int[2];
// test[0] = 21; tes [1] = 42;
// Un appel à PrintArray(test) affiche:
42, 21
```

Tips

Notez qu'il n'y a pas de virgule à la suite du dernier élément, par contre un retour à la ligne est présent. Pour afficher du texte sans retour à la ligne, vous pouvez vous renseignez sur Console.Write.

Square

Ecrivez une méthode qui affiche un carrée à l'écran, pour cela, deux paramètres sont donnés, la largeure du carré ainsi que le caractère composant Une case.

Prototype

```
static void DrawSqare(int Width, char element);
```

Examples

```
//Un appel à DrawSqare(4, '*') devrait produire
****
****
****
****
```

Un peu d'algorithmie

Ces exercies peuvent sembler complexes, si vous rencontrez des soucis, n'hésitez pas à appeler un assitant ou à passer à la partie suivante.

Happy

En mathématique, un entier naturel est un nombre heureux si, lorsqu'on calcule la somme des carrés de ses chiffres qui le compose, la somme des carrés des chiffres du nombre obtenu et ainsi de suite, on aboutit au nombre 1.

```
Par example
1² + 9² = 1 + 81 = 82
8² + 2² = 64 + 4 = 68
6² + 8² = 36 + 64 = 100
1² + 0² + 0² = 1 + 0 + 0 = 1
```

Vous devez donc écrire une fonction qui prend un entier en paramètre, et renvoie un booléen indiquand si le nombre est un nombre heureux

Prototype

```
static bool Happy(int number);
```

Bonus

Ecrivez une fonction qui detecte les nombres malheureux. Un nombre malheureux est un nombre dont la procédure utilisé pour trouver un nombre heureux le fait arriver sur l'un des nombres suivants: 4, 16, 37, 58, 89, 145, 42 et 20. Une fois arrivé sur l'un de ces nombres, la procédure boucle indéfiniment.

Pascal

Le triangle de Pascal est un tableau triangulaire, qui permet de retrouver les coefficients binomiaux. Le coefficient de la ligne i et de la colonne j est la somme des coefficients situés à (i-1, j) et (i-1, j-1).

Vous devez écrire une fonction qui affiche un triange de Pascal de profondeur n

Prototype

```
static void Pascal(int n);
```

Example

```
//Un appel à Pascal(5) produit
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

OOP

Pour introduire légérement la programation objet, nous allons réaliser une arène, dans lasquelle divers champions vont s'affronter. Nous allons tout d'abord modéliser un combattant standard, puis définir des combattants spécialisés à l'aide de l'héritage. Enfin, nous nous attacherons à définir une arène.

Fighter

Rempailleur de chaises LVL 1

Pour notre premier combattant, nous allons tout d'abord commencer par définir son constructeur. Un combattant classique possède un nom, une force ainsi qu'un nombre de point de vie.

```
class Fighter
{

    // Nous définissons tout les champs de notre champion avec une visibilité
    // protected, afin d'empécher toutes modifications par une source externe,
    // nous allons voir par la suite comment premettre la lecture de ces
```

```
    // éléments.

    protected string Name;
    protected int Health;
    protected int Strength;

    // Ici nous définissons le contructeur, vu que seul le nom est passé,
    // nous vous laissons définir la force ansi que le nombre de PV par les
    // valeurs qui vous semblent correcte. Attention, si ces champs ne sont
    // pas initialisés, ils auront comme valeur par défaut 0.

    public Fighter(string Name)
    {
        //FIXME
    }
}
```

Say my name !

Nous venons de définir la base de notre combattant, mais il lui manque de nombreuses méthodes pour être utilisable, nous allons ici définir des méthodes classiques pour obtenir des informations sur une instance. Ces fonctions ne devraient pas vous poser trop de problèmes.

```
class Fighter
{
    // Constructeur et Champs

    // Cette méthode comme son nom l'indique retourne le nom du combattant
    public string GetName()
    {
        //FIXME
    }

    // Indique si le combattant est mort, c'est à dire si son nombre de point
    // de vie est inférieure ou égale à zero.
    public bool IsDead()
    {
        //FIXME
    }
}
```

Ces méthodes que nous venons d'implémentez sont ce que l'on appelle des 'Getter', c'est à dire des méthodes qui n'ont pour but que de donner des informations

sur l'objet. A l'inverse il existe des 'Setter' qui n'ont pour rôle que de donner une valeur à un champs de l'objet, tout en maitrisant l'accès à ce dernier.

Il s'avère que ce procédé est très répandu, et C# offre un moyen plus lisible de faire ce travail au travers d'attributs. Nous n'allons pas entrer dans les détails aujourd'hui mais si vous êtes curieux n'hésitez pas à poser une question à un assitant ou à vous documenter sur internet.

Virtual, j'invoque ton nom

Notre combattant est déjà un peu plus présentable, mais il lui manque les moyens d'attaquer et de se défendre. Nous allons nous y atteler tout de suite.

Les méthodes que nous implémentons sont définies comme virtuelle, car nous voulons pouvoir les redéfinir dans les classes filles.

```
class Fighter
{
    // Constructeur, champs et méthodes

    // Cette méthode affiche le cri de guerre de notre guerrier, et renvoie
    // le nombre de dommage effectué.
    public virtual int Attack()
    {
        //FIXME
    }

    // Cette méthode est appelé quand notre combattant est attaqué, on récupère
    // les dommages de l'ennemi grace à sa méthode Attack, puis l'on applique
    // ces dégats à notre héros (this). L'ajout d'un message lors de la mort
    // de la pauvre cible est bienvenue.
    publique virtual void OnAttack(Fighter Ennemy)
    {
        //FIXME
    }
}
```

Le TP le plus classe du monde

Maintenant que nous avons l'ossature d'un combattant, nous pouvons la spécialiser afin de varier nos attaques et nos méthodes de défenses.

Chaussette

Nous allons ici définir un barbare, un vrai, avec des muscles partout ainsi qu'un sens de la stratégie limité. Ainsi, nous n'allons pas changer ses méthodes d'attaque et de défense, mais nous allons booster ses stats.

```
class Barbarian : Fighter
{
    // Ici, nous définissons le constructeur de notre barbare, il doit faire
    // appel au constructeur du Fighter (voir les slides), puis appliquer un
    // boost de 50% sur la force ainsi que sur le nombre de PVs.
    public Barbarian(string Name);
}
```

Magicien, spécialité debug

Ici, nous allons travailler sur un mage, ce dernier possède un nombre de points de mana limités que nous pouvons utiliser pour réaliser de fortes attaques

```
class Magician : Fighter
{
    protected int mana;

    // Ici, nous construisons notre magicien, tout en initialisant sa valeur
    // de mana. 5 semble être un bonne valeur.
    public Magician(string Name);


    // L'attaque d'un mage est particulière: si il le peut, il utilise l'un
    // de ses points de manas pour doubler son attaque, sinon il n'attaque
    // qu'avec la moitié de sa force. Encore une fois, un message de
    // circonstance peut être afficher.
    public override int Attack()
    {
        //FIXME
    }
}
```

Voleur, par ce qu'il en faut un

De par leur nature, les voleurs sont des gens fourbes. Ainsi, ils ont une attaque plus faible que la moyenne, mais ils sont capables des coups les plus retords.

```
class Thief : Fighter
{
    // Encore une fois, nous redefinissons notre constructeur, mais dans le cas
    // d'un voleur, nous devons faire attention à ce que sa vie soit diminuée
    // de 33% pour représenter sa faiblesse
    public Thief(string Name);

    // Le voleur à la capacité, avec suffisamment de chance, de tricher et de
    // réduire automatiquement la vie de son ennemi à zero. Pour cela, il
    // profite dur fait que le champ Health ai une visibilité protected, ce qui
    // permet à un autre objet Fighter (ou l'une de ses classe fille) d'y avoir
    // accès.
    // Pour représenter la moule karmique du voleur, nous vous invitons à
    // vous documenter sur la classe Random.
    // Bien entendu, dans le cas ou le voleur manque de chance, il se prend
    // l'attaque de son ennemi.
    public override void OnAttack(Figher Ennemy)
    {
        //FIXME
    }
}
```

L'arène

Maintenant que nous avons des combattans digne des plus grandes sagas héroiques, nous allons les faires combattre. Pour cela nous allons créer une classe Arena, mais nous n'allons pas lui fournir de constructeur ou de méthode standard, nous n'allons utiliser que des méthodes statiques.

```
class Arena
{


    // Cette méthode fait combattre entre eux les deux combattant jusqu'à
    // la mort d'un d'entre eux. Une annonce du champion à la fin du combat
    // serait du plus bel effet.
    static void Fight(Fighter f1, Fighter f2)
    {
        //FIXME
    }


    // Cette méthode redéfinit fight pour faire combattre deux équipes de
    // combattants. Pour cela nous choisissons un guerrier dans chaque équipe,
    // puis nous les faisons se battre à mort. Nous continuons ainsi jusqu'a
```

```
    // ce que tout une équipe soit décimée.
    static void Fight(Figher[] t1, Fighter[] t2)
    {
        //FIXME
    }
}
```

Pour finir

Voilà, nous en avons finis avec le TP, vous pouvez au choix ajouter de nouveaux types de combattans, ou encore élargir leurs capacités en ajoutant plus de méthodes virtuelles à la classe Fighter (par exemple, apporter secours à un allié). Si vous cherchez plus de défi, n'hésitez toujours pas à appeler un assitant.