TP 8 : AsciiDots

Consignes de rendu

À la fin de ce TP, vous devrez rendre un dépôt Git respectant l'architecture suivante :

```
csharp-tp8-prenom.nom/
|-- README
|-- .gitignore
|-- AsciiDot/
    |-- AsciiDot.sln
    |- AsciiDot/
        |-- AsciiDot.cs
        |-- AsciiDot.csproj
        |-- Board.cs
        |-- Direction.cs
        |-- Dot.cs
        |-- Memory.cs
        |-- Point.cs
        |-- Program.cs
        |-- Token/
            |-- Lexer.cs
            |-- Token.cs
            |-- TokenChar.cs
            |-- TokenConditional.cs
            |-- TokenDuplicate.cs
            |-- TokenEmpty.cs
            |-- TokenEnd.cs
            |-- TokenInput.cs
            |-- TokenInsertor.cs
            |-- TokenMirror.cs
            |-- TokenNumber.cs
            |-- TokenOperator.cs
            |-- TokenOutput.cs
            |-- TokenPath.cs
            |-- TokenQuote.cs
            |-- TokenReflector.cs
            |-- TokenStart.cs
            |-- TokenValue.cs
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Remplacez `prenom.nom` par votre propre login.
— Le fichier `README` est obligatoire.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— Retirez tous les tests de votre code.
— **Le code doit compiler !**

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

Table des matières

1 Cours

1.1 Objectifs

Ce TP a pour objectif de mettre en pratique les notions de la programmation objet orienté abordés dans le TP4 et dans le TP6 concernant les classes, les objets, l'abtraction et l'héritage. Vous êtes invités à revoir les notions qui sont présentés dans ces TPs.

1.2 Programmation en orienté objet

1.2.1 Polymorphisme par héritage

Lorsqu'une classe concrète (non abstraite) est destinée à être une classe mère dans un contexte d'héritage, les méthodes concernées par l'héritage doivent être définies avec le mot-clé `virtual`.

Contrairement aux méthodes asbtraites ou aux attributs abstraits qui doivent être nécessairement implémentés dans les classes filles ce n'est maintenant plus obligatoire dans la mesure où une implémentation existe déjà dans la classe mère.

Lorsqu'on souhaite redéfinir dans une sous-classe une méthode ou un attribut virtuel, on utilise le mot-clé `override`. Si on souhaite appeler la méthode définie dans la classe mère lors de la redéfinition, on utilise le mot-clé `base`.

Considérons l'exemple dedans :

```
1  public class Vec2
2  {
3      public double X;
4      public double Y;
5
6      public Vec2(double x, double y)
7      {
8          this.X = x;
9          this.Y = y;
10     }
11
12     public virtual double Norm()
13     {
14         return Math.Sqrt(X * X + Y * Y);
15     }
16 }
17
18 public class Vec3 : Vec2
19 {
20     public double Z;
21
22     public Vec3(double x, double y, double z) : base(x, y)
23     {
24         this.Z = z;
25     }
26
27     public override double Norm()
28     {
29         return Math.Sqrt(Math.Pow(base.Norm(), 2) + Z * Z);
30     }
31 }
```

Dans l'exemple ci-dessus, la classe `Vec2` décrit un vecteur du plan et permet de calculer sa norme grâce à la méthode `Norm` (notez le mot-clé `virtual`). La classe `Vec3` hérite de la classe `Vec2` et décrit un vecteur de l'espace. Elle redéfinit la méthode `Norm` avec le mot-clé `override`.

Pour l'exemple, nous appelons la méthode `Norm` de la classe mère avec `base.Norm()`. Notez que dans la pratique, ce n'est pas le meilleur moyen de le faire…

1.3 Surcharge

Une surcharge est le fait de créer des méthodes dans la même classe ayant le même nom et le même type de retour mais avec des paramètres différents. Cette technique permet notamment d'appeler une méthode suivant les paramètres qu'on souhaite lui attribuer afin d'adopter le comportement défini. C'est ainsi qu'on surcharge la méthode.

```
public static void Hello()
{
    Console.WriteLine("Hello!");
}

public static void Hello(String name)
{
    Console.WriteLine("Hello {0}!", name);
}

public static void Main(string[] args)
{
    Hello();
    Hello("Sherlock");
}
```

On a en sortie :

```
    Hello!
    Hello Sherlock!
```

La surcharge peut s'appliquer sur des constructeurs :

```
public class Clock
{
    public int Hours;
    public int Minutes;
    public int Seconds;

    public Clock()
    {
        this.Hours = 0;
        this.Minutes = 0;
        this.Seconds = 0;
    }

    public Clock(int hours, int minutes, int seconds)
    {
        this.Hours = hours;
        this.Minutes = minutes;
        this.Seconds = seconds;
    }
}

public static void Main(string[] args)
{
    Clock clock = Clock();
    Clock deadline = Clock(23, 42, 00);
}
```

1.4 Gestion des exceptions

1.4.1 Qu'est qu'une exception ?

Lors de l'exécution d'un programme, il peut être sujet à des cas d'erreurs qui bloquent son fonctionnement. On a la possibilité de prévenir c'est-à-dire de détecter et de réparer certains de ces cas particuliers. On introduit pour cela la notion d'*exceptions*.

On pose dans le programme des conditions exceptionnelles et selon l'exception appelée, on applique un traitement pour pouvoir la gérer et poursuivre (ou non) le déroulement du programme en cours d'execution.

Par exemple, l'une des exceptions les plus fréquentes lors des opérations arithmétiques est la division par zéro.

En C#, les exceptions sont levées avec le mot-clé `throw`. Pour la gestion d'exception, on utilise les blocs `try`, `catch` et `finally`. Ils comportent un corps c'est-à-dire un bloc de code fermé par des accolades.

Le bloc `try` permet d'essayer d'exécuter des instructions qui peuvent potentiellement échouer et renvoyer une exception. Le bloc `catch` permet de capturer un type d'exception qu'on indique entre parenthèses. Enfin, le bloc optionnel `finally` sera toujours exécuté.

Lorsqu'une exception se produit, elle est propagée dans la pile d'appels jusqu'à rencontrer un `catch`. Si une exception n'est pas « attrapée », elle cause l'arrêt du programme.

```
1  public static void PrintDivision(int a, int b)
2  {
3      try
4      {
5          Console.WriteLine("{0} / {1} = {2}", a, b, a / b);
6      }
7      catch (DivideByZeroException)
8      {
9          Console.Error.WriteLine("You can't divide by zero!");
10     }
11 }
12
13 public static void Main(string[] args)
14 {
15     PrintDivision(15, 3);
16     PrintDivision(15, 0);
17 }
```

En exécutant le programme ci-dessus, on obtient :

```
1  15 / 3 = 5
2  You can't divide by zero!
```

Il est également possible de créer ses propres exceptions. Il suffit pour cela de créer une classe héritant de `Exception` :

```
1  public class EditorException : Exception
2  {
3      public EditorException() : base()
4      {
5      }
6
7      public EditorException(string msg) : base(msg)
8      {
9      }
10 }
11
12 public static void PrintEditor(string editor)
13 {
14     if (editor == "vim")
15         throw new EditorExpcetion("Maybe you should try emacs...");
16     if (editor == "emacs")
17         throw new EditorExpcetion("Try to find an editor that isn't only used " +
18                                   "to play Tetris...");
19     Console.WriteLine("{0}? Wise choice!", editor);
20 }
```

Notons l'usage du mot-clé `throw` pour lever une exception. Vous remarquerez d'ailleurs que le constructeur de la classe `EditorException` est surchargé.

2 Exercices

Important

Tous les caractères ont leur valeur **Unicode** en hexadecimal spécifiée à côté entre parenthèses. Vous pouvez récupérer le caractère correspondant grâce à la fonction Python `chr()`.

Exemple :

```
>>> chr(0x2022)
'•'
```

2.1 Lore

Sherlock Holmes a encore besoin de vous pour sa dernière enquête. En recherchant un criminel, il est tombé à plusieurs reprises sur des pages entières contenant des suites de caractères a priori incompréhensibles. En voici l'une d'elles :

```
  /---$_">"-*---~-$#-&
  | /--;---\| [!]-\
  | *------++--*#1/
  | | /1#\ ||
 [*]*{-}-*~<+*?#-.
  *--------+-</
  \-#0----/
```

Après quelques recherches, il a découvert un vieil ouvrage mentionnant un langage qui lui semblait fort similaire : l'AsciiDots.

2.2 L'AsciiDots

L'AsciiDots est un langage de programmation en deux dimensions. Le principe est simple : des points (dots) se déplacent en suivant des chemins tout en exécutant différentes actions selon les caractères qu'ils rencontrent. Pas de panique ! Les prochaines parties expliquent le principe plus en détail.

Pas la référence

Vous pouvez expérimenter avec le langage sur ce site :

asciidots.herokuapp.com

Attention : ce site/programme ne peut pas être considéré comme une référence pour ce projet.

2.2.1 Dot

Le caractère `.` ($0x2e$) sert à créer un nouveau `dot`. Il est également possible d'utiliser le point centrale • ($0x2022$).

Code minimaliste :

```
.
```

2.2.2 Chemin

Les points peuvent se déplacer horizontalement sur les tirets `-` ($0x2d$), verticalement sur les barres verticales `|` ($0x7c$) et dans les deux sur les plus `+` ($0x2b$).

Lorsqu'un point sort du chemin, il tombe dans le vide et est détruit. Quand il n'existe plus de point, le programme s'arrête.

Au début de la partie, le point s'oriente dans la direction du chemin le plus proche.

Exemple avec un chemin horizontal :

```
.-----
```

Il est également possible de créer plusieurs points les uns à la suite des autres.

```
.--.-----
```

Dans le cas où il y a plusieurs chemins possibles, la direction est déterminée selon le sens horaire : $\uparrow$, $\rightarrow$, $\downarrow$ puis $\leftarrow$.

Exemple

```
  |
-.-
  |
```

Dans cet exemple, le point part vers le haut.

2.2.3 The End

Un autre moyen de finir le programe est qu'un point arrive sur une esperluette `&` ($0x26$).

Exemple

```
.--&----
.-----
```

Dans cet exemple, le point du bas n'arrivera jamais à la fin du chemin car il sera arrêté avant par l'autre au moment où celui-ci arrivera sur le `&`. Le point du haut n'atteindra évidemment pas non plus la fin du chemin.

2.2.4 Miroir, miroir, mon beau miroir...

On peut déjà faire beaucoup, mais on ne peut que se déplacer que dans une seule direction. Le premier moyen de changer de direction est de rencontrer un miroir `\` ($0x5c$) ou `/` ($0x2f$). Comme son nom l'indique, lorsqu'un point arrive dessus, celui-ci est réfléchi, dans une direction orthogonale.

Exemple

```
      |
      |
.---/
```

Le point commence sa course vers la droite, avant de rencontrer le miroir, qui va le faire repartir vers le haut.

NB : On peut utiliser les deux côtés du miroir.

```
    .
    |
    |
----\----
    |
    |
    .
```

Le point du haut va se retrouver à droite tandis que celui du bas va finir sa course à gauche.

Il est maintenant possible de réaliser nos premières boucles infinies [1]

```
/---.--\
|      |
|      |
\------/
```

NB : Quand un point rencontre un point de départ `.`, celui-ci est considéré comme un chemin multi-direction `+`. Ce sera le même comportement pour tout autre caractère qui ne change pas la direction du point.

Les dots qui viennent de la gauche ont le même comportement dans ces 3 programmes :

```
   |
.--+--
   |
   .
```

```
   |
.--.--
   |
   .
```

```
   |
.--P--
   |
   .
```

(Ici la lettre `P` est considérée comme un `+`)

2.2.5 Inserteur

Un nouveau problème se pose maintenant : on est incapable d'insérer un point dans un chemin. Pas de panique ! Il existe pour cela les inserteurs :

— vers le haut `^` ($0x5e$)
— vers la droite `>` ($0x3e$)
— vers le bas `v` ($0x76$)
— vers le gauche `<` ($0x3c$)

Tout point arrivant perpendiculairement à un inserteur sera redirigé dans le sens correspondant à celui-ci.

1. Merci de ne pas en abuser… [2]
2. Voir [1]

Exemple

```
.----->---
      |
      .
```

Le point du bas sera inséré sur le chemin horizontal et finira sa course à droite.

NB : Un inserteur n'a aucun effet sur un point arrivant à contre-sens :

```
.----<---
```

Dans ce code, le point va ignorer l'inserteur et parcourir tout le chemin horizontal.

2.2.6 Réflecteur

On a maintenant presque tous les mouvements. Il nous manque juste la possibilité de rebondir ou d'être réfléchi. Pas de panique il existe 2 symboles pour ça :

— `(` ($0x28$) : reflète un dot venant de la droite.
— `)` ($0x29$) : reflète un dot venant de la gauche.

Il n'existe malheureusement pas de version pour réfléchir vers le haut et le bas.

Exemple :

```
(.-)
```

Ce code est la plus petite boucle infinie possible [1].

Le point va commencer sa course vers la droite puis va rencontrer le premier réflecteur et va repartir dans l'autre sens (vers la gauche). Celui-ci va continuer dans l'autre sens jusqu'à rencontrer le deuxième réflecteur qui va le renvoyer vers la droite. Et ainsi de suite juste qu'à la fin de l'éternité.

2.2.7 Duplicateur

Que se passe-t-il si l'on souhaite dupliquer un `dot` ? Il existe pour cela un token spécifique : le duplicateur `*` ($0x2a$). Lorsque qu'un `dot` arrive dessus, il est dupliqué en trois exemplaires : dans la direction actuelle ainsi que dans les deux directions orthogonales.

Exemple

```
      &
      |
      |
.---*--&
      |
      |
      &
```

Dans cet exemple, au moment où le point arrivant de la gauche rencontre le duplicateur `*`, il est multiplié dans trois directions : à droite, en haut et en bas.

2.2.8 Affectation

C'est bien beau de pouvoir se déplacer, mais il serait appréciable de pouvoir stocker des valeurs dans nos points. Il existe pour cela le croisillon (et pas « *sharp* »...) `#` ($0x23$).

Chaque `dot` contient en effet une valeur numérique. Celle-ci vaut par défaut 0 lorsqu'un `dot` est créé.

Exemple

```
.---#42---&
```

Dans cet exemple, le `dot` contient initialement la valeur `0`, puis, après avoir rencontré les caractères `#42`, la valeur `42`.

Récupérer une valeur sur l'entrée standard Nous avons vu qu'il était possible d'affecter une valeur numérique à un `dot`. En AsciiDot, il existe également la possibilité de demander une valeur à l'utilisateur. On utilise pour cela `#?`. Lorsqu'un `dot` rencontre `#?`, il reste sur le point d'interrogation dans l'attente d'une valeur sur l'entrée standard. Si la chaîne donnée par l'utilisateur n'est pas numérique, on considère que la valeur `0` a été rentrée.

2.2.9 Affichage - Dessine-moi un mouton

Lorsque l'on veut afficher un message à nos utilisateurs, il existe pour cela le dollar `$` ($0x24$) qui est par défaut équivalent à `Console.WriteLine`.

Il existe différents cas en fonction de ce qui suit le `$` :

- S'il est suivi d'une chaîne de caractères entre guillements doubles `"` ou simples `'`, la chaîne est affichée suivie d'un retour à la ligne.
- S'il est suivi de `#`, le valeur numérique contenue dans le `dot` sera affichée suivie d'un retour à la ligne.
- S'il est suivi de `a`, le comportement dépend de ce qui suit :
  - S'il est suivi d'une chaîne, le comportement est identique au cas où il n'y a que la chaîne.
  - S'il est suivi de `#`, le caractère associé au code Unicode contenu dans le `dot` sera affiché suivi d'un retour à la ligne.
- S'il est suivi d'un nombre strictement positif de `_`, le comportement est le même que dans les cas décrits ci-dessus à l'exception qu'il n'y a pas de retour à la ligne.
- Dans tous les autres cas, rien ne se passe.

Attention

L'ordre des caractères `a` et `_` lorsqu'ils suivent un `$` sont interchangeables.

Exemple

```
.---$_"Hello, World"-#33-$a#--#2---$_"4"-$#--&
```

Sortie :

```
Hello, World!
42
```

Note

Dans la vraie version de l'AsciiDots, il existe une différence entre `"` et `'`. En effet, dans le premier cas, la chaîne ne sera affichée qu'à la fin alors que les caractères sont affichés au fur et à mesure dans le deuxième. Nous considèrerons ici que `'` se comporte comme `"`.

2.2.10 Opérateurs

Que se passe-t-il si l'on souhaite effectuer des opérations entre les `dots` ? Il existe pour cela deux syntaxes : `[op]` et `{op}`. « `op` » désigne un caractère parmi la liste des opérateurs valides en AsciiDots :

- $*$ ($0x2a$), la multiplication
- $/$ ($0x2f$) ou $\div$ ($0xf7$), la division
- $+$ ($0x2b$), l'addition
- $-$ ($0x2d$), la soustraction
- $\%$ ($0x25$), le modulo
- $\hat{}$ ($0x5e$), le passage à la puissance
- $\&$ ($0x26$), le `ET` logique
- o ($0x6f$), le `OU` logique
- x ($0x78$), le `XOR` logique
- $>$ ($0x3e$), strictement supérieur
- $\geq$ ($0x2265$), supérieur ou égal
- $<$ ($0x3c$), strictement inférieur
- $\leq$ ($0x2264$), inférieur ou égal
- $=$ ($0x3d$), test d'égalité
- $\neq$ ($0x2260$), test de non égalité

Note sur le NOT

En AsciiDots, il existe également l'opérateur `!` qui correspond au `NON` logique unaire. Pour des raisons de simplicité, nous décidons de ne pas l'inclure et de nous limiter aux opérateurs binaires.

Note sur les opérateurs booléens

En AsciiDots, les booléens `vrai` et `faux` sont représentés par les entiers `1` et `0` respectivement. Le résultat de l'opération `42 > 11` sera donc `1`.

Le comportement des opérateurs binaires peut être déroutant. Lorsqu'un `dot` rencontre un `[op]` ou `{op}`, il y reste jusqu'à ce qu'un autre `dot` arrive dans une direction orthogonale. Lorsque deux `dots` arrivant de deux directions orthogonales se trouvent alors sur un même opérateur, les étapes suivantes dépendent du type d'opérateur en question :

- S'ils se trouvent sur un opérateur délimité par des crochets tel que `[/]`, l'opération s'effectue entre le `dot` arrivant **verticalement** et le `dot` arrivant **horizontalement**, dans cet ordre. Le `dot` résultant, portant le résultat de l'opération, sortira dans le direction du `dot` arrivé verticalement. Le `dot` arrivé horizontalement est détruit.
- S'ils se trouvent sur un opérateur délimité par des accolades tel que `{/}`, l'opération s'effectue entre le `dot` arrivant **horizontalement** et le `dot` arrivant **verticalement**, dans cet ordre. Le `dot` résultant, portant le résultat de l'opération, sortira dans le direction du `dot` arrivé horizontalement. Le `dot` arrivé verticalement est détruit.

Un petit exemple pour clairifier tout ça

```
          .
          |
          #
          2
          6
          |
.-#100--[+]  .-#3-\
          |       |
          \-------{/}---$#--&
```

Sortie :

```
42.0
```

Détaillons cet exemple. Le dot le plus en haut commence par descendre, prendre la valeur `26` et attendre un camarade sur le `+`. Arrive ensuite le `dot` le plus à gauche qui prend la valeur `100` et arrive sur le `+` sur lequel se trouve déjà un autre `dot`. L'opération `26 + 100` est réalisée. Le `dot` sortant par le bas porte ainsi la valeur `126`.

Pendant ce temps, un `dot` plus à droite a pris la valeur `3` et attend un autre `dot` arrivant horizontalement sur le `/`. Lorsque le `dot` porteur de la valeur `126` arrive par la gauche sur le `/`, l'opération `126 / 3` est réalisée. Le `dot` sortant par la droite porte ainsi la valeur `42.0`.

Cette valeur est finalement affichée sur la sortie standard.

2.2.11 Contôle de flux

Tout langage qui se respecte doit avoir une structure conditionnelle. L'AsciiDots étant évidemment un langage qui se respecte, il possède un équivalent : le tilde `~` ($0x7e$).

Lorsqu'un `dot` arrive sur un `~`, il attend un autre `dot` arrivant d'une direction orthogonale. Lorsque deux `dots` se trouvent simultanément sur un `~`, le `dot` arrivant **verticalement** détermine la direction de sortie de celui arrivant **horizontalement**.

Si le `dot` arrivant verticalement a une valeur différente de `0`, alors le `dot` arrivé horizontalement sort par le haut. Sinon, le `dot` arrivé horizontalement continue selon sa direction initiale.

Dans tous les cas, le `dot` arrivé verticalement est détruit et le `dot` arrivé horizontalement conserve sa valeur.

Exemple

```
        /--$"true"--$#-&
        |
.-#42-~--$"false"-$#-&
        |
.-#1--/
```

Sortie :

```
true
42
```

Dans cet exemple, le point le plus en haut commence par prendre la valeur `42` et attendre un autre point sur le `~`. Pendant ce temps, le point du bas prend la valeur `1` et arrive sur le `~` sur lequel se trouve déjà un autre `dot`. Le `dot` arrivé verticalement contenant la valeur `1` (et $1 \neq 0$), le point arrivé horizontalement sort par le haut.

2.3 La boîte à outils de Bob

Avant d'implémenter toutes les règles de l'AsciiDots nous allons implémenter 3 classes :
— `Direction`
— `Point`
— `Dot`

2.3.1 Direction

Info

Toutes les fonctions sont à faire dans `Direction.cs`.

`Direction` est une énumération contenent les quatre directions cardinales : une énumération est juste une table clé-valeur. De ce fait nous avons numeroté les directions dans le sens horaire (cela vous sera utile plus tard).
— `Up` : 0
— `Left` : 1
— `Down` : 2
— `Right` : 3

Rappels – Arithmetic go brrr

Comme les énumérations peuvent être considérées comme des entiers, vous pouvez faire de l'arithmetique dessus :

```
1
2  enum Fruit {
3    Apple = 0,
4    Banana = 1,
5    Grape = 2,
6    Watermelons = 3,
7    Rasberry = 4,
8  }
9
10 static void Main()
11 {
12     Fruite myFavoriteFruit = Fruit.Apple;
13     Fruite anotherFruit = (Fruit) (((int) myFavoriteFruit + 2) * 2);
14     Console.WriteLine(anotherFruit)
15 }
```

Ce qui donne :

```
1  Fruit.Rasberry
```

Comme les énumérations ne peuvent pas contenir de méthodes, nous allons les implémenter dans la classe statique `DirUtils`.

SameAxis Vous devez implémenter la fonction `SameAxis` qui vérifie si les deux directions qui lui sont passées sont colinéaires.

```
1  public static bool SameAxis(Direction d1, Direction d2);
```

Paramètres
— `d1` la première direction
— `d2` la deuxième direction

Valeur de retour Vrai si `d1` et `d2` sont sur le même axe. Dans les autre cas, elle renvoie faux.

Exemple :

```
1  Console.WriteLine(DirUtils.SameAxis(Direction.Up, Direction.Down));
2  Console.WriteLine(DirUtils.SameAxis(Direction.Up, Direction.Left));
```

donnera :

```
1  True
2  False
```

Rotate

```
1  public static Direction Rotate(Direction direction);
```

Effectue une rotation dans le sens horaire.

Paramètre
— `direction` la direction à tourner.

Valeur de retour Renvoie la direction une fois la rotation dans le sens horaire effectuée.

Exemple :

```
1  Console.WriteLine(DirUtils.Rotate(Direction.Up));
2  Console.WriteLine(DirUtils.Rotation(Direction.Left));
```

donnera :

```
1  Direction.Right
2  Direction.Up
```

Invert

```
1  public static Direction Invert(Direction direction);
```

Donne la direction opposée à celle donnée.

Paramètre
— `direction` : la direction à inverser

Valeur de retour Renvoie la direction opposée à `direction`.

Exemple :

```
1 Console.WriteLine(DirUtils.Invert(Direction.Up));
2 Console.WriteLine(DirUtils.Invert(Direction.Left));
```

donnera :

```
1 Direction.Down
2 Direction.Right
```

DeltaX

```
1 public static int DeltaX(Direction direction);
```

Donne la composante horizontale de la direction.

Paramètre
— `direction` la direction dont on doit trouver la composante X.

Valeur de retour Renvoie la composante horizontale de la direction.

Exemple :

```
1 Console.WriteLine(DirUtils.DeltaX(Direction.Up));
2 Console.WriteLine(DirUtils.DeltaX(Direction.Left));
```

donnera :

```
1 0
2 -1
```

DeltaY

```
1 public static int DeltaY(Direction direction);
```

Donne la composante verticale de la direction.

Paramètre
— `direction` la direction dont on doit trouve la composante Y.

Valeur de retour Renvoie la composante verticale de la direction.

Exemple :

```
1 Console.WriteLine(DirUtils.DeltaY(Direction.Up));
2 Console.WriteLine(DirUtils.DeltaY(Direction.Left));
```

donnera :

```
1 1
2 0
```

2.3.2 Point

Pour localiser des éléments sur la carte (que ce soit des cases ou des `dots`), il nous faut un moyen de représenter des points. La classe `Point` est là pour ça.

Info

Toutes les fonctions sont à faire dans `Point.cs`.

Constructeurs Vous allez voir votre premier exemple de surcharge.

Vous devez implémenter les deux constructeurs suivant qui initialiseront les attributs `X` et `Y` du point.

```
1 public Point(int x, int y);
2 public Point(Point point);
```

Tips

Un constructeur qui prend une instance de la classe pour la copier est appelé constructeur de copie (oui, c'est très original…).

Clone

```
1 public Point Clone();
```

Maintenant que vous avez savez ce qu'est un constructeur de copie vous allez pouvoir implémenter la methode `Clone`, qui…clone le points. C'est une autre manière de créer des constructeurs de copie.

Pouquoi utiliser `Clone` quand on a un constructeur de copie, me diriez-vous ?

C'est très simple, c'est une question de goût et de propreté.

Exemple :

```
1 new MyClass((new MyClass(myPoint)).doSomeThing(ref result));
2 // est équivalent à :
3 myPoint.Clone()
4         .doSomeThing(ref result)
5         .Clone();
```

ACDC Tips

Si vous voulez garder votre code propre il y a raccourcis spécial vous permettant de le formater automatiquement (et le rendre lisible pour vos ACDC) :

Ctrl + Alt + S

Step

```
1 protected Point Step(Direction direction);
```

Déplace le point dans la direction donnée en paramètre.

Paramètres
— `direction` la direction dans laquelle se déplace le point.

Valeur de retour Renvoie le point lui-même (cela permet de chaîner les modifications).

MoveTo

```
1  public Point MoveTo(Direction direction);
```

Récupère un **nouveau point** qui correspond au déplacement du point dans la direction donnée.

Paramètre
— `direction` la direction dans laquelle se déplace le point.

Valeur de retour Renvoie un nouveau point résultant du déplacement.

2.4 Dot

2.4.1 Dot

C'est la principale composante de nos programmes, il dérive de `Point`.

Info

Toutes les fonctions sont à faire dans `Dot.cs`.

Constructeurs Vous allez commencer par implémenter les deux constucteurs de `Dot` :

```
1  public Dot(int x, int y, Direction direction);
2  public Dot(Point point, Direction direction);
```

Step

```
1  public void Step();
```

Cette fonction est une surcharge de la méthode héritée de `Point`. Elle ne prend pas de paramètres, contrairement à sa petite sœur. Elle fait avancer le `dot` d'un pas.

2.4.2 Memory

La classe `Memory` permet de ne pas à avoir à retenir des informations concernant l'environnement dans la classe `Dot`.

Info

Toutes les fonctions sont à faire dans `Memory.cs`.

Constructeur Vous commencerez par implémenter les deux constructeurs de `Memory` :

```
1  public Memory();
2  public Memory(Memory memory);
```

Vous aurez pour cela besoin d'utiliser l'enumération `Environment`, qui permet de donner une information sur le contexte actuel d'un `dot`. L'environnement doit être initialisé à la valeur par défaut.

Flush

```
1  public void Flush();
```

La méthode `Flush` permet de récupérer les valeurs actuellement présentes dans la file et déterminer quoi en faire. Deux types d'actions peuvent être réalisées :

- Si le premier caractère de la file est `$`, il faut appliquer les règles présentées dans la section « Récupérer une valeur sur l'entrée standard », puis vider la file.
- Si le premier caractère de la file est `#`, il faut appliquer les règles présentées dans la section « Exemple », puis vider la file.

Dans tous les autres cas, il ne faut rien faire.

Vous aurez pour cela besoin d'implémenter les méthodes suivantes :

```
1  public void Assignment(string str);
2  public void Display(string str);
```

La méthode `Assignment` prend une chaîne commençant par `#` et applique les règles correspondantes pour l'afficher sur la sortie standard. `Display` a un fonctionnement analogue : elle prend une chaîne commençant par `$` et applique les règles relatives à l'affichage.

Attention

Il faut bien penser à gérer tous les cas ! Cela inclut notamment les exemples suivants :

```
1  var memory = new Memory();
2
3  memory.Display("$a_\"Votai Test.\""); // Votai Test.
4  memory.Display("$$$_____\"Votai Test.\""); // Votai Test.
5  memory.Display("$__aa__\"Votai Test.\""); // Votai Test.
```

2.5 Board

Dans cette partie, nous allons transformer les fichiers de programme en `Board`. Nous allons donc implémenter le parsing de fichier.

Axes et Directions Tout le programme sera stocké dans une matrice `Matrix`. Cette matrice contient dans sa première dimension les colonnes, et dans sa deuxième dimension ses lignes. Pour ce qui est du sens des axe nous mettrons le sens positif des ordonnées vers le haut et le sens positif des abscisses à droite, le zéro se trouvant dans le coin inférieur gauche (comme les graphiques en mathématiques).

Vous pouvez utiliser les méthodes `Get` et `Set` si vous avez des doutes sur le sens des axes.

Une méthode `PrintString` vous est fournie.

Info

Toutes les fonctions sont à faire dans `Board.cs`.

Constructeur

```
1  public Board(string path);
```

Vous devez dans un premier temps créer le constructeur de `Board` (il est conseillé d'implementer la fonction `LoadContent`).

Cette fonction se décompose en quatre étapes :

1. Récupérer le contenu du fichier
2. Le découper en lignes et en colonnes (attention, toutes les lignes ne sont pas forcément de la même longueur)
3. Calculer les dimensions de la matrice
4. Parser toutes les cases du tableau avec la fonction qui vous est fournie dans `Token/Lexer.cs` :

```
1  public static Token Lex(char[][] table, int x, int y);
```

2.6 Tokens

`Lexer.Lex` vous renvoie des tokens. Vous pouvez voir la liste de tous les tokens à implémenter dans `Lexer.TokenTypes` et `DirectedTokenType`. Les fonctions de lexing vous sont données. Il vous suffit de penser à décommenter les différentes lignes quand vous avez fini de créer/implémenter les différents tokens.

Vous pouvez jeter un coup d'œil aux méthodes de la classe `Lexer`. Pas de panique, si vous ne les comprenez pas, sachez juste qu'elles permettent de tester tous les différents tokens possibles jusqu'à tomber sur le bon. De plus elles gèrent la direction des sorties s'il y en a besoin (cela sera pratique pour certains tokens que vous implémenterez plus tard).

Nous vous invitons à aller lire l'implémentation de la classe `Token` où son utilisation et fonctionnement sont expliqués en détail.

Vous avez aussi deux exemples d'implémentation de cette classe abstraite :

1. `TokenEmpty` : Qui correspond au blanc/espace dans la grille
2. `TokenStart` : Qui sont les points de départ des `dot`.

Info

Toutes les fonctions sont à faire dans le dossier `Token`.

Dans le dossier `Token`, vous devez créer les fichiers suivants contenant les classes de même nom héritant toutes de `Token` :

- `TokenEnd.cs`
- `TokenPath.cs`
- `TokenMirror.cs`
- `TokenInsertor.cs`
- `TokenReflector.cs`
- `TokenChar.cs`
- `TokenNumber.cs`, dans lequel la classe `TokenNumber` hérite de `TokenChar`
- `TokenQuote.cs`
- `TokenOutput.cs`
- `TokenValue.cs`
- `TokenInput.cs`
- `TokenDuplicate.cs`
- `TokenConditional.cs`

Attention !

Faites attention à bien nommer les fichiers et les classes correspondantes.

Pour chacune de ces classes, vous devez implémenter un constructeur de la forme :

```
1  public Token(char c);
```

où `Token` est évidemment à remplacer par le nom de chaque classe.

Attention

Pensez bien à appeler le constructeur de la classe parent.

Vous devez également implémenter l'attribut `AllowedChars`, héritant de celui de la classe `Token`, contenant les caractères acceptés pour le token :

```
1  protected override string AllowedChars;
```

Pour chaque classe, il faut implémenter la méthode `Update`.

```
1  protected override bool Update(Dot dot);
```

Cette méthode détermine en fonction du caractère actuel et de l'environnement le nouvel environnement et les actions éventuelles à effectuer. Elle retourne un booléen indiquant si il y a une action à réaliser.

Important

Un point important est la mise à jour de la liste `Queue` du `dot`. En effet, lorsque le `dot` se trouve dans un environnement différent de `Environment.Default`, elle contient tous les caractères rencontrés depuis le dernier moment où l'environnement était `Environment.Default`.

Exemple :

```
1  .--$"Hello, World!"-&
```

Ici, lorsque l'on rencontre `$`, on commence à ajouter des `tokens` dans `Queue` et l'environnement est défini à `Environment.Output`. Lorsque l'on rencontre le premier `"`, l'environnement devient `Environment.DoubleQuote` et on continue de rajouter les `tokens` dans `Queue` lorsqu'on les rencontre.
Ainsi, au moment où le `dot` est sur le caractère `W`, la file contient `$"Hello, W`.
Lorsque le `dot` rencontre le guillement final, après avoir ajouté le `token`, la méthode `Flush` du `dot` est appelée, vidant ainsi `Queue`. L'environnement redevient `Environment.Default`.

Enfin, il faut implémenter dans chaque fichier la méthode `Action`, héritant là-encore de la classe parent :

```
1  protected override List<Dot> Action(Dot dot);
```

Cette méthode, executée lorsque `Update` renvoie `true`, modifie si besoin la direction du `dot`. C'est le cas par exemple pour des `tokens` tels que le miroir ou l'inserteur.

De plus, elle renvoie une liste contenant le ou les éventuels nouveaux `dots` une fois l'action effectuée. Si le `dot` est encore en vie, il doit être contenu dans cette liste (voir l'implémentation par défaut dans `Token.cs`).

Conseil 1

Vous êtes libres de rajouter tous les attributs que vous considérez utiles dans les classes que vous créez.

Conseil 2

Il peut arriver pour certains `tokens` que les fonctions `Update` et `Action` n'ajoutent pas de comportement par rapport à la classe parent. Il n'est pas nécessaires dans ces cas-là de les implémenter dans les classes filles.

Opérateurs

Les `tokens` représentants les opérateurs sont un peu particuliers. Commencez par créer le fichier `Token/TokenOperator.cs` contenant la classe `TokenOperator`, héritant de `Token`. Une fois cette classe créée, pensez bien à décommenter la ligne correspondant dans le fichier `Lexer.cs`.

Attributs La classe `TokenOperator` doit contenir les attributs privés suivants :

```
1  private readonly List<Dot> _dotQueue;
2  private Direction _direction;
```

L'attribut `_direction` vaudra `Direction.Up` dans le cas d'un opérateur entre crochets tel que `[+]` et `Direction.Right` dans le cas d'un opérateur entre accolades tel que `{+}`.

L'attribut `_dotQueue` contient l'ensemble des `dots` arrivés sur l'opérateur et attendant un autre `dot` dans une direction orthogonale, dans leur ordre d'arrivée.

Conseil

N'hésitez pas à relire la partie expliquant le fonctionnement des opérateurs.

Vous devez également implémenter l'attribut `AllowedChars` contenant tous les opérateurs possibles : « `*/÷+-%^&ox> < =` ».

Constructeur Vous devez implémenter le constructeur de la classe (pensez bien à appeler le constructeur de la classe parent) :

```
1  public TokenOperator(char c, Direction direction);
```

Compute Vous devez maintenant implémenter la méthode `Compute` :

```
1  private double Compute(double lhs, double rhs);
```

En fonction de l'opérateur, cette méthode effectue l'opération appropriée et retourne le résultat. Pour les comparaisons, une différence dans l'intervale $[0; 0.001[$ sera ignorée.

Conseil

Les erreurs d'arithmétique telles que la division par zéro ne seront pas testées.

Action Enfin, vous devez écrire la méthode `Action`, héritant de `Token` :

```
protected override List<Dot> Action(Dot dot);
```

Vous devez ici implémenter les comportements relatifs aux opérateurs. N'hésitez pas, si besoin, à vous référer à la section correspondante.

2.7 Pas à pas

Maintenant que nous avons notre `board` et nos tokens fonctionelles, il faut faire bouger ces dots.

2.7.1 Au commencement, il y avait des Dots

Info

Les fonctions suivantes sont à faire dans `Board.cs`.

Au debut d'un programme, nous devons trouver tous les points de `board` et leur direction de départ.

Pour cela vous devez implémenter la méthode :

```
public List<Dot> StartDots();
```

Cette méthode crée la liste de tous les dots créé au début du programme orienté dans la bonne direction (Voir 2.2.2).

Valeur de retour Renvoie la liste de `dots` trouvés dans la matrice.

Il vous est vivement conseillé d'implémenter les méthodes suivantes :

```
private static bool IsStartToken(Direction startDirection, Token.Token token);
private bool FindStartDirection(Dot dot);
```

2.7.2 Let's play

Info

Les fonctions suivantes sont à faire dans `AsciiDot.cs`.

`AsciiDot` est une classe pour exécuter des programmes.

Constructeur Dans un premier temps vous allez implémenter le constructeur qui initialise `AsciiDot`. Pensez bien à intialiser la liste des dots de départ.

```
public AsciiDot(Board board);
```

UpdateDot Vous allez implémenter maintenant la méthode `UpdateDot` qui applique à un `dot` l'action de la case sur laquelle il se trouve. Celle-ci retourne les dots de génération suivante résultant de l'application de la case.

```
1  private List<Dot> UpdateDot(Dot dot);
```

Paramètre
— `dot` le dot à appliquer les actions

Valeur de retour Retourne la liste de dots de la génération suivante.

Info

Pour appliquer l'action de la case, il faut appeler la méthode `public List<Dot> Apply(Dot dot)` fournie dans le fichier `Token.cs`. Cette méthode met à jour l'environnement du dot et applique l'action de la case au dot.

UpdateGame Met à jour la liste `dots` c'est-à-dire les `dots` en vie de `board` et crée la prochaine génération de `dots`.

```
1  public void UpdateGame();
```

Launch La fonction `Launch` démarre le programme et continue l'exécution tant que le programme n'est pas finie.

Si le booléen `print` passé en paramètre de la fonction est vrai, vous devez afficher `board` (`ToString` peux être très pratique) et attendre $100ms$ à chaque itération du programe.

```
1  public void Launch(bool print);
```

Paramètre
— `print` booléen pour l'affichage de `board`

Info

Le programme continue lorqu'il y existe encore des `dots` en vie.

**There is nothing more deceptive,
than an obvious fact.**