TP 8 : AsciiDots

Submission instructions

At the end of the practical, your Git repository must follow this architecture:

```
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
```

Do not forget to check the following requirements before submitting your work:

- You shall obviously replace `firstname.lastname` with your login.
- The `README` file is mandatory.
- There must be no `bin` or `obj` folder in the repository.
- You must respect the prototypes of the given and asked functions.

- Remove all personal tests from your code.
- **The code MUST compile !**

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

Contents

1 Courses

1.1 Objectives

The goal of this tutorial is to put into practice the notions of object-oriented programming discussed in praticals 4 and 6 concerning classes, objects, abtraction and inheritance. You are encouraged to review the notions that are presented in these tutorials.

1.2 Object-oriented programming

1.2.1 Polymorphism by inheritance

When a concrete (not abstract) class is intended to be a parent class in an inheritance context, the methods concerned by the inheritance must be defined with the keyword `virtual`.

Unlike abstract methods or attributes. Which always have to be implemented in the child class, this is no longer compulsory as long as an implementation already exists in the parent class.

When we want to redefine a method or a virtual attribute in a subclass, we use the keyword `override`. If we want to call a method defined in the parent class at the time of redefinition, we use the keyword `base`.

Let us see the example below:

```
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 }
```

In the example given, the class `Vec2` describes a vector of the plane and allows the user to calculate their norm using the method `Norm` (note the keyword `virtual`). The class `Vec3` inherits from the class `Vec2` and describes a vector of the space. It redefines the method `Norm` with the keyword `override`.

For example, we call the method `Norm` of the parent class with `base.Norm()`. Note that in practice, this is not the best way to do it.

1.3 Overload

An overload is the creation of methods in the same class with the same name and the same return type but with different parameters. This technique allows you to call a method according to the parameters that we want to assign to it in order to adopt the defined behavior. This is how we overload a method.

```
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");
}
```

We get the following output:

```
Hello!
Hello Sherlock!
```

The overload can be applied to constructors:

```
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 Exception handling

1.4.1 What is an exeption?

During the execution of a program, there may be some errors that prevent it from running properly. It is possible to predict, i.e. to detect and repair some of these particular cases. This is why, we introduce the notion of *exceptions*.

We set exceptional conditions in the program and according to the exception called, we use a proccess to be able to manage and to continue (or not) the execution of the program in progress.

For example, one of the most common exceptions for arithmetic operations is division by zero.

In C#, exceptions are thrown with the keyword `throw`. The blocks `try`, `catch` and `finally` are used for exception handling. They have a body, i.e. a block of code closed by curly brackets.

The `try` block allows us to try executing instructions that may potentially fail and return an exception. The `catch` block allows us to catch a type of exception that is indicated between parentheses. Finally, the optional block `finally` will always be excecuted.

When an exception is thrown, it is propagated through the call stack until it meets a `catch` block. If an exception is not "catch", it causes the end of the program.

```
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 }
```

We get the following output:

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

It is also possible to create your own exceptions. To do this, you just need to create a class inheriting from `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 not used only " +
18                                   "to play Tetris...");
19     Console.WriteLine("{0}? Wise choice!", editor);
20 }
```

Note the presence of the keyword `throw` to throw an exception. You may notice that the constructor of the `EditorException` class is overloaded.

2 Exercices

Important

All character as their **Unicode** value specify in hexadecimal after in parentheses.
You can get the corresponding character with the `chr()` python function.

Example:

```
>>> chr(0x2022)
'•'
```

2.1 Lore

Sherlock Holmes needs you again for his last investigation. While tracking down for a criminal, several times he has found documents containing sequences of characters that are incomprehensible at first glance. Here is one of them:

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

After mutch research, he discovered an old book mentioning a language that was very similar to this: AsciiDots.

2.2 AsciiDots

AsciiDots is a two-dimensional programming language. The principe is simple: the points (`dots`) move along paths and excecute different actions according to the characters they meet. Do not worry! The next parts detail the principle.

Not the reference

You can experiment with the language on this site:

asciidots.herokuapp.com

Warning this site/program can't be consider as a reference for this project.

2.2.1 Dot

The character `.` (unicode $0x2e$) is used to create a new `dot`. It is also possible to use the central point • ($0xe280a2$).

Minimalist code:

```
.
```

2.2.2 Path

The `dots` can move horizontally on the dashes `-` ($0x2d$), vertically on the vertical bar `|` ($0x7c$) and in both on the plus `+` ($0x2b$).

When a `dot` goes out of a path, it falls into the void and is destroyed. When there are no more points, the program stops.

At the beginning of the game, the `dot` is oriented in the direction of the nearest path.

Example with a horizontal path:

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

It is also possible to create multiple `dots` one after another.

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

In the case where there are multiple possible paths, the direction is determined clockwise: ↑, →, ↓ and ←.

Example:

```
  |
-.-
  |
```

In this example, the `dot` goes up.

2.2.3 The End

A program finish when there are no more moving dots on the board.

An alternative way to finish the program is that a `dot` arrives on an ampersand `&` ($0x26$).

Example:

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

In this example, the `dot` at the bottom will never reach the end of the path because it will be stopped beforehand by the other one when it arrives on the `&`. The `dot` at the top will obviously not reach the end of the path either.

2.2.4 Mirror, mirror, mirror on the wall, ...

We can already do a lot, but we can only move in one direction. The first way to change direction is to encounter a mirror `\` ($0x5c$) or `/` ($0x2f$). As the name suggests, when a `dot` hits it, it is reflected, in an orthogonal direction.

Example:

```
1       |
2       |
3   .---/
```

The `dot` starts to move to the right, before it meets the mirror, which will make it go back up.
NB: Both sides of the mirror can be used.

```
1       .
2       |
3       |
4   ----\----
5       |
6       |
7       .
```

The `dot` at the top goes to the right whereas the `dot` at the bottom ends up on the left.

It is now possible to realize our first infinite loops. [1]

```
1   /---.--\
2   |      |
3   |      |
4   \------/
```

NB: When a `dot` meets a starting point `.`, this is considered as a multi-directional path `+`. This will be the same behavior for any other character that does not change the direction of the dot.
The dots comming from the left and the bottom as the same behavous in each of this 3 programs:

```
1       |
2   .--+--
3       |
4       .
```

```
1       |
2   .--.--
3       |
4       .
```

```
1       |
2   .--P--
3       |
4       .
```

(Here the letter `P` is consider as a `+`)

[1] Thanks for not abusing it...[2]
[2] See [1]

2.2.5 Insertor

A new problem arises now: we are not capable to insert a `dot` in a path. Do not panic! There are insertors for that:

- upwards `^` ($0x5e$)
- to the right `>` ($0x3e$)
- downwards `v` ($0x76$)
- to the left `<` ($0x3c$)

Any `dot` coming perpendicularly to an insertor will be redirected in the direction corresponding to it.

Example:

```
1  .----->---
2         |
3         .
```

The `dot` at the bottom will be inserted on the horizontal path and will finish its course to the right.

NB: An insertor has no effect on a `dot` coming from an the opposite direction:

```
1  .----<---
```

In this code, the `dot` will ignore the insertor and travel the entire horizontal path.

2.2.6 Reflector

We now have almost all the moves. We just lack the ability to bounce or to be reflective. Don't panic, there are 2 symbols for that:

- `(` ($0x28$): reflect dots that come from the right.
- `)` ($0x29$): reflect dots that come from the left.

Unfortunately there is no version for up and down.

Example

```
1  (.-)
```

This code is the smallest possible infinite loop [1].

The point will begin its course towards the right then will meet the first reflector and will continue in the other direction (towards the left). it will continue in the other direction until it meets the second reflector which will reflect to the right. And so on, until the end of eternity.

2.2.7 Duplicator

What happens if we want to duplicate a `dot`? There is a specific token for this: the duplicator `*` ($0x2a$). When a `dot` comes on it, it is duplicated in three copies: in the current direction and in the two orthogonal directions.

Example:

```
1       &
2       |
3       |
4   .---*--&
5       |
6       |
7       &
```

In this example, when the `dot` coming from the left meets the duplicator `*`, it is duplicated in three directions: right, up and down.

2.2.8 Assignment

It's great to be able to move around, but it would be good to be able to store values in our dots. For this purpose, there is the hashtag `#` ($0x23$).

Each `dot` contains a numerical value. This value is set to 0 by default when a `dot` is created.

Example:

```
1   .---#42---&
```

In this example, the `dot` initially contains the value `0`, then, after meeting the characters `#42`, the value `42`.

Retrieve a value from the standard input: We have seen that it is possible to assign a numerical value to a `dot`. In AsciiDot, it is also possible to ask a value from the user. We use for this `#?`. When a `dot` meets `#?`, it stays on the question mark waiting for a value on the standard input. If the string given by the user is not numerical, we consider that the value `0` has been entrered.

2.2.9 Display - Draw me a sheep

When we want to display a message to our users, there is the dollar sign `$` ($0x24$), which is by default equivalent to `Console.WriteLine`.

There are different cases depending on what follows the dollar sign:

- If it is followed by a string delimited by either `"` or `'`, the string is dislayed followed by a line break.
- If it is followed by `#`, the numerical value contained in the `dot` will be displayed followed by a line break.
- If it is followed by `a`, the behavior depends on what follows:
  - If it is followed by a string, the behavior is identical to the case where there is only the string.
  - If it is followed by `#`, the character associated with the Unicode code contained in the `dot` will be displayed followed by a line break.
- If it is followed at least by one `_`, the behavior is the same as in the cases described above except that there is no line break.

- In all the other cases, nothing happens.

Caution

The order of the characters `a` and `_` when they follow a `$` are interchangeable.

Example:

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

Output:

```
Hello, World!
42
```

Note

In the real version of AsciiDots, there is a difference between `"` and `'`. Indeed, in the first case, the string will be displayed only at the end, whereas the characters are displayed progressively in the second. We will consider here that `'` behaves like `"`.

2.2.10 Operators

What happens if we want to perform operations between the `dots`? There are two syntaxes for this: `[op]` and `{op}`. "`op`" designates any character among the list of valid operators in AsciiDots:

- $*$ ($0x2a$), multiplication
- $/$ ($0x2f$) or $\div$ ($0xf7$), division
- $+$ ($0x2b$), addition
- $-$ ($0x2d$), subtraction
- $\%$ ($0x25$), modulus
- $\hat{}$ ($0x5e$), exponent
- $\&$ ($0x26$), logical `AND`
- o ($0x6f$), logical `OR`
- x ($0x78$), logical `XOR`
- $>$ ($0x3e$), greater than
- $\geq$ ($0x2265$), greater than or equal
- $<$ ($0x3c$), less than
- $\leq$ ($0x2264$), less than or equal
- $=$ ($0x3d$), equal
- $\neq$ ($0x2260$), not equal

Note on the NOT

In AsciiDots, there is also the `!` operator which corresponds to the logical unary `NOT`. For simplicity, we decide not to include it and to limit ourselves to binary operators.

Note on boolean operators

In AsciiDots, the booleans `true` and `false` are represented by the integers `1` and `0` respectively. The result of the operation `42 > 11` will be therefore `1`.
For logical boolean operator (`OR`, `AND` and `XOR`) any value that is not `0` is constider as `true` (only `0` is consider as `false`). The result of the operation `42 o 0` will be therefore `1`.

The behavior of binary operators can be confusing. When a `dot` encounters a `[op]` or `{op}`, it stays there until another `dot` arrives from an orthogonal direction. When two `dots` arriving from two orthogonal directions are then on the same operator, the next steps depend on the type of operator in question:

- If they are on a bracketed operator such as `[/]`, the operation is performed between the `dot` arriving **vertically** and the `dot` arriving **horizontally**, in that order. The resulting `dot`, carrying the result of the operation, will go out in the direction of the `dot` arriving vertically. The horizontally arrived `dot` is destroyed.
- If they are on a bracketed operator such as `{/}`, the operation is performed between the `dot` arriving **horizontally** and the `dot` arriving **vertically**, in that order. The resulting `dot`, carrying the result of the operation, will go out in the direction of the `dot` arriving horizontally. The vertically arrived `dot` is destroyed.

A small example to illustrate this

```
1            .
2            |
3            #
4            2
5            6
6            |
7   .-#100--[+]   .-#3-\
8            |         |
9            \-------{/}---$#--&
```

Output:

```
1  42.0
```

Let us detail this example. The highest `dot` starts by going down, taking the value `26` and waiting for an other dot to arrive at the `+`. Then comes the leftmost `dot` which takes the value `100` and arrives on the `+` on which another `dot` is already. The operation `26 + 100` is performed. The `dot` coming out of the bottom has the value `126`.

Meanwhile, a `dot` further to the right has taken the value `3` and is waiting for another `dot` arriving horizontally on the `/`. When the `dot` carrying the value `126` arrives by the left on the `/`, the operation `126 / 3` is performed. The `dot` going out by the right thus has the value `42.0`.

This value is finally displayed on the standard output.

2.2.11 Flow control

Any respectable language must have a conditional structure. AsciiDots being obviously a respectable language, it has an equivalent structure: the tilde `~` ($0x7e$).

When a `dot` arrives on a `~`, it waits for another `dot` arriving from an orthogonal direction. When two `dots` are simultaneously on a `~`, the `dot` arriving **vertically** determines the exit direction of the one arriving **horizontally**.

If the vertically arriving `dot` has a value different from `0`, then the horizontally arriving `dot` exits from the top. Otherwise, the horizontally arriving `dot` continues in its original direction.

In all cases, the `dot` that arrived vertically is destroyed and the `dot` that arrived horizontally keeps its value.

Example:

```
1        /--$"true"--$#-&
2        |
3  .-#42-~~~$"false"-$#-&
4        |
5  .-#1--/
```

Output:

```
1    true
2    42
```

In this example, the topmost point starts by taking the value `42` and waiting for another point on the `~`. Meanwhile, the bottom point takes the value `1` and arrives on the `~` on which is already another `dot`. The vertically arrived `dot` contains the value `1` (and $1 \neq 0$),and the horizontally arrived point leaves by the top.

2.3 Bob's toolbox

Before implementing all the rules of the AsciiDots we will implement three classes:

- `Direction`
- `Point`
- `Dot`

2.3.1 Direction

Info

All functions will be implemented in `Direction.cs`.

`Direction` is an enumeration containing the four cardinal directions: an enumeration is just a key-value table. Therefore, we have enumerated the directions in a clockwise direction (this will be useful later).

- `Up`: 0
- `Left`: 1
- `Down`: 2
- `Right`: 3

Reminders – Arithmetic go brrr

Since enumerations can be considered as integers, you can do arithmetic on them:

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

This gives:

```
1  Fruit.Rasberry
```

As enumerations cannot contain methods, we will implement them in the static class `DirUtils`.

SameAxis You must implement the function `SameAxis` which checks if the two directions passed are collinear.

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

Parameters

- `d1` the first direction
- `d2` the second direction

Return value True if `d1` and `d2` are on the same axis. Otherwise, it returns false.

Example

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

will give:

```
1  True
2  False
```

Rotate

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

Rotates clockwise.

Parameter

- `direction` the direction to turn.

Return value Return the direction after the clockwise rotation.

Example

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

will give:

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

Invert

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

Gives the opposite direction to the one given.

Parameter

- `direction` the direction to be reversed

Return value Return the opposite direction to `direction`.

Example

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

will give:

```
Direction.Down
Direction.Right
```

DeltaX

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

Gives the horizontal component of the direction.

Parameter

- `direction` the direction whose X component is to be found.

Return value Return the horizontal component of the direction.

Example

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

will give:

```
0
-1
```

DeltaY

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

Gives the vertical component of the direction.

Parameter

- `direction` the direction whose Y component is to be found.

Return value Return the vertical component of the direction.

Example

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

will give:

```
1
0
```

2.3.2 Point

In order to locate elements on the `board` (whether they are cases or `dots`), we need a way to represent position. The `Point` class is there for that.

Info

All functions will be implemented in `Point.cs`.

Constructors You are going to see your first example of overloading.

You must implement the following two constructors which will initialize the attributes `X` and `Y` of the point.

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

Tips

A constructor that takes an instance of the class to copy it is called copy constructor (yes, it's very original...).

Clone

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

Now that you know what a copy constructor is you can implement the `Clone` method, which... clones the points. This is another way to create copy constructors.

Why use `Clone` when you have a copy constructor, you might ask?

It's very simple, it's a question of taste and cleanliness.

Example:

```
1 new MyClass((new MyClass(myPoint)).doSomeThing(ref result));
2 // is equivalent to:
3 myPoint.Clone()
4         .doSomeThing(ref result)
5         .Clone();
```

ACDC Tips

If you want to keep your code clean there are special shortcuts that allow you to automatically format it (and make it readable for your ACDC):

Ctrl + Alt + S

Step

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

Moves the point in the direction given in parameter.

Parametors

- `direction` the direction in which the point moves.

Return value Return the point itself (this allows to chain the modifications).

MoveTo

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

Retrieves a **new point** that corresponds to the displacement of the point in the given direction.

Parameter

- `direction` the direction in which the point moves.

Return value Return a new point resulting from the move.

2.4 Dot

2.4.1 Dot

It is the main component of our programs, it derives from `Point`.

Info

All functions will be implemented in `Dot.cs`.

Constructors You will start by implementing the two constuctors of `Dot`:

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

Step

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

This function is an overload of the method inherited from `Point`. It does not take any parameters, unlike its parrent constructor. It moves the `dot` forward by one step.

2.4.2 Memory

The `Memory` class makes it possible not to have to retain information concerning the environment in the `Dot` class.

Info

All functions will be implemented in `Memory.cs`.

Constructor You will start by implementing the two constructors of `Memory`:

```
public Memory();
public Memory(Memory memory);
```

To do this, you will need to use the enumeration `Environment`, which allows you to give information about the current context of a `dot`. The environment must be initialized to the `Default` value.

Flush

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

The `Flush` method is used to retrieve the values currently in the queue and determine what to do with them. Two types of actions can be performed:

- If the first character in the queue is `$`, you must apply the rules presented in the section "Retrieve a value from the standard input:", then empty the queue.
- If the first character in the queue is `#`, you must apply the rules presented in the section "Assignment", then empty the queue.

In all other cases, nothing should be done.

For this you will need to implement the following methods:

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

The `Assignment` method takes a string starting with `#` and applies the corresponding rules to display it on the standard output. `Display` works in a similar way: it takes a string beginning with `$` and applies the display rules.

Attention

You have to think about how to handle all cases! This includes the following examples:

```
var memory = new Memory();

memory.Display("$a_\"Votai Test.\""); // Votai Test.
memory.Display("$$$_____\"Votai Test.\""); // Votai Test.
memory.Display("$__aa__\"Votai Test.\""); // Votai Test.
```

2.5 Board

In this part, we will transform the program files into `Board`. We will thus implement the file parsing.

Axes and Directions Whole the program will be stored in a matrix `Matrix`. This matrix contains in its first dimension the columns, and in its second dimension the rows. For the direction of the axes we will put the positive direction of the y-axis upwards and the positive direction of the x-axis to the right, with the zero in the lower left corner (like the graphs in mathematics).

You can use the `Get` and `Set` methods if you have doubts about the direction of the axes.

A method `PrintString` is provided.

Info

All functions will be implemented in `Board.cs`.

Constructor

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

You will in a first time create the `Board` constructer (it's highly recommended to implement the `LoadContent` function).

This function is divided into four steps:

1. Load the contents of the file
2. Cut it into rows and columns (note that not all rows are the same length)
3. Determine the dimensions of the matrix
4. Parsing all the cases in the table with the function provided below in `Token/Lexer.cs` (you don't have to deal with any errors management):

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

2.6 Tokens

`Lexer.Lex` returns tokens. You can see the list of all the tokens to implement in `Lexer.TokenTypes` and `DirectedTokenType`. The lexing functions are given to you. You just have to remember to uncomment the different lines when you have finished creating/implementing the different tokens.

You can take a look at the methods of the `Lexer` class. Do not worry, if you don't understand, just know that they allow you to test all the different possible tokens until you find the right one. Moreover, they manage the direction of the outputs if necessary (this will be useful for some tokens that you will implement later).

We invite you to read the implementation of the `Token` class where its use and functioning are explained in detail.

You also have two examples of implementation of this abstract class:

1. `TokenEmpty`: corresponds to the white/space in the matrix
2. `TokenStart`: are the starting points of the `dot`.

Info

All functions will be implemented in `Token` directory and will take the same name as the token class. example: `TokenEnd` will be implemented in the `Token/TokenEnd.cs` File and will be obvously public.

In the `Token` folder, you must create the following files containing classes of the same name, all inheriting from `Token` :

- `TokenEnd.cs`
- `TokenPath.cs`
- `TokenMirror.cs`
- `TokenInsertor.cs`
- `TokenReflector.cs`

- `TokenChar.cs`
- `TokenNumber.cs`, in which `TokenNumber` inherits `TokenChar`
- `TokenQuote.cs`
- `TokenOutput.cs`
- `TokenValue.cs`
- `TokenInput.cs`
- `TokenDuplicate.cs`
- `TokenConditional.cs`

Caution !

Be careful to name the files and the corresponding classes properly.

For each of these classes, you must implement a constructor of the form:

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

where `Token` should obvously be replaced by the name of each class.

Caution

Remember to call the constructor of the parent class.

You must also implement the attribute `AllowedChars`, inheriting from that of the class `Token`, containing the characters accepted for the token:

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

For each class, the `Update` method must be implemented.

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

This method determines, according to the current character and the environment, the new environment and the potential actions to perform. It returns a boolean indicating whether is an action to perform.

Important

An important point is the update of the attribute `Queue` of the `dot`. Indeed, when the `dot` is in an environment different from `Environment.Default`, it contains all the characters encountered since the last time the environment was `Environment.Default`.

Example :

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

Here, when we meet `$`, we start adding `tokens` in `Queue` and the environment is defined at `Environment.Output`. When we meet the first `"`, the environment becomes `Environment.DoubleQuote` and we continue to add the `tokens` in `Queue` when we meet them. Thus, when the `dot` is on the character `W`, the file contains `$"Hello, W`.
When the `dot` meets the closing quote, after having added the `token`, the `Flush` method of the `dot` is called, thus emptying the `Queue`. The environment becomes `Environment.Default` again.

Finally, the `Action` method must be implemented in each file, again inheriting from the parent class:

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

This method, executed when `Update` returns `true`, modifies if necessary the direction of the `dot`. This is the case for example for `tokens` such as the mirror or the inserter.

Moreover, it returns a list containing the possible new `dots` once the action is performed. If the `dot` is still alive, it must be contained in this list (see the default implementation in `Token.cs`).

Tip 1

You are free to add any attributes that you consider useful in the classes you create.

Tip 2

It can happen for some `tokens` that the functions `Update` and `Action` do not add any behavior compared to the parent class. In these cases, it is not necessary to implement them in the child classes.

Operators

The `tokens` representing the operators are a bit special. Start by creating the file `Token/TokenOperator.cs` containing the `TokenOperator` class, inheriting from `Token`. Once this class has been created, remember to uncomment the corresponding line in the file `Lexer.cs`.

Attributes The class `TokenOperator` must contain the following private attributes:

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

The attribute `_direction` will be equal to `Direction.Up` in the case of an operator between brackets such as `[+]` and `Direction.Right` in the case of an operator between curly braces such as `{+}`.

The attribute `_dotQueue` contains the set of `dots` arrived on the operator and waiting for another `dot` coming from an orthogonal direction, in their order of arrival.

Tip

Do not hesitate to reread the part explaining how operators work.

You must also implement the attribute `AllowedChars` containing all possible operators: "`*/÷+-%^&ox> < =`".

Constructor You must implement the constructor of the class (remember to call the constructor of the parent class):

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

Compute You must now write the method `Compute`:

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

Depending on the operator, this method performs the appropriate operation and returns the result. For comparisons, a difference in the interval $[0; 0.001[$ will be ignored.

Tip

Arithmetic errors such as divisions by zero will not be tested.

Action Finally, you must write the method `Action`, inheriting from `Token`:

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

You must implement the behaviours of the operators. Do not hesitate, if you feel the need, to read the corresponding section.

2.7 Step by step

Now that we have our `board` and our functional tokens, we have to make these dots move.

2.7.1 In the beginning, there were Dots

Info

All functions will be implemented in `Board.cs`.

At the beginning of a program, we have to find all the points of `board` and their starting direction. For this you need to implement:

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

This method creates the list of all the dots created at the beginning of the program oriented in the right direction (See 2.2.2).

Return value Return a list of `dots` found in Matrix.
You are strongly advised to implement the following methods:

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

2.7.2 Let's play

Info

All functions will be implemented in `AsciiDot.cs`.

`AsciiDot` is a class for running programs.

Constructor First you will implement the constructor which initializes `AsciiDot`. Don't forget to initialize the list of starting dots.

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

UpdateDot You are now going to implement the method `UpdateDot` which applies to a `dot` the action of the box on which it is located. This method returns the next generation dots resulting from the application of the square.

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

Parameter

- `dot` the dot to apply the actions

Return value Returns the list of dots of the next generation.

Info

To apply the action of the case, you must call the method `public List<Dot> Apply(Dot dot)` provided in the file `Token.cs`. This method updates the environment of the dot and applies the action of the case to the dot.

UpdateGame Updates the `dots` list, i.e. the `dots` alive in the `board`, and creates the next generation of `dots`.

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

Launch The function `Launch` starts the program and continues the execution until the program is finished.

If the boolean `print` passed in parameter of the function is true, you must display `board` (`ToString` can be very practical) and wait $100ms$ at each iteration of the program.

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

Parameter

- `print` boolean for the display of `board`

Info

The program continue while there are moving `dots`.

There is nothing more deceptive,
than an obvious fact.