# Projet Scala S1

## Présentation

Le but du projet de semestre est de construire un algorithme de dessin, permettant de créer diverses formes géométriques dans la console.

Le programme marche de la façon suivante :

1. La *canvas* (i.e. la *toile* de dessin) est affichée
2. Le programme demande l'action à exécuter
3. L'utilisateur écrit une commande composée d'une *action* et éventuellement d'*arguments*(exemple: "`draw_line 0,1 2,5 x`") - Ici l'action est `draw_line`, et les arguments sont `0,1 - 2,5 - x`
4. Le programme applique l'action sur la canvas et retourne à l'étape 1.

La base du programme est fournie dans le fichier `Main.scala` (permettant de gérer la boucle d'exécution, la lecture de l'*input* utilisateur, …), votre rôle consiste à implémenter les différentes actions.

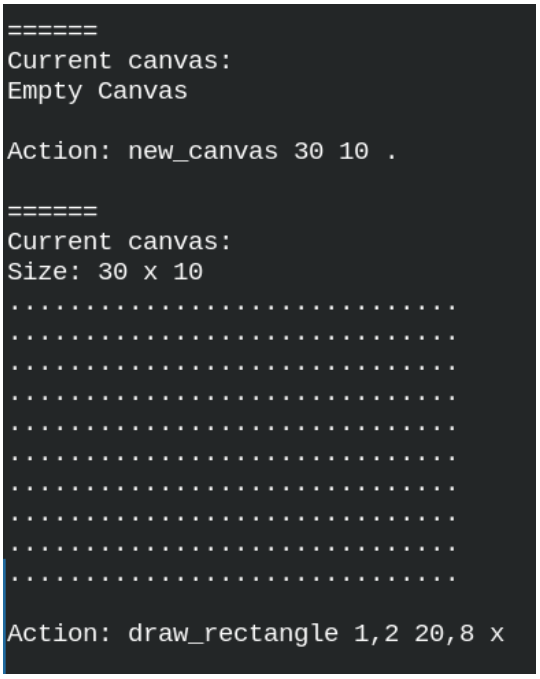
Le programme est déjà exécutable mais ne peut rien faire, hormis s'arrêter avec l'action `exit` qui permet de quitter le programme.

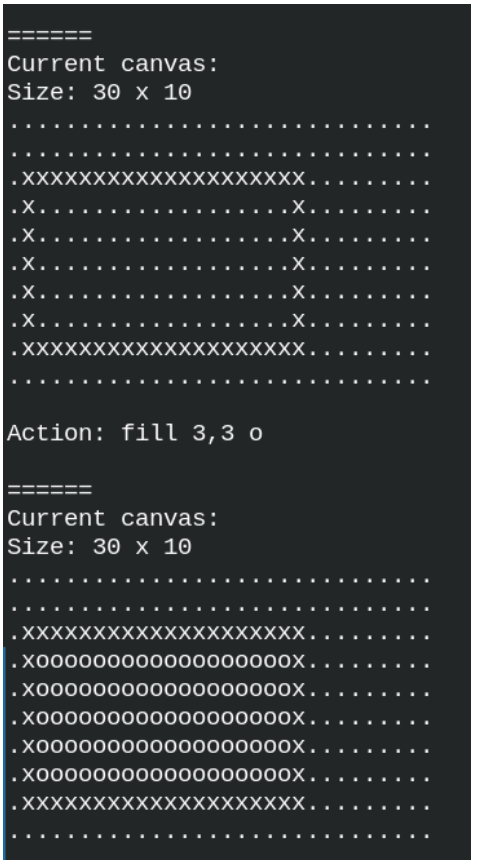
Pour lancer le programme :

- Compiler le fichier : `scalac Main.scala`
- L'exécuter : `scala Main`

```
Starting application

======
Canvas:
Empty canvas

Action: draw_line 0,0 10,10 x
ERROR: An error occured. Unkonown command

======
Canvas:
Empty canvas

Action: exit

Exiting application
Final status Received exit signal
```

Une fois les diverses actions implémentée, en les appelant successivement il sera possible de dessiner dans la canvas :




# Organisation

Le projet est à réaliser par équipe de **2 à 3 personnes**.

**Un fichier `README.md` est attendu contenant au minimum :**

- **Les instructions pour lancer le programme**
- **La liste des différentes actions implémentées avec les arguments attendus, ce qu'elles font, les limites connues**
- **Une explication des différents choix techniques fait si cela semble pertinent**
- **Tout autre information qui vous semble intéressante (les bugs connus, les problèmes rencontrés, etc.)**
- **Le partage des tâches**

Le fichier `Main.scala` modifiée ainsi que les autres fichiers scala que vous avez pu écrire (si nécessaire)

**Attention**, quelques contraintes :

- Le code doit compilé et être exécutable, même si tout n'a pas été implémenté (si vous avez essayer une action mais qu'elle ne fonctionne pas, n'hésitez pas à la commenter pour la laisser présente sans qu'elle casse le programme)
- Aucune variable / objet mutable ne doit être présent

**Date de rendu** : Vendredi 17 mars à 18h00. Tout retard sera pénalisé.

# À implémenter

*Vous n'avez pas besoin d'éditer la fonction run, ni la case class Status, sauf si cela vous semble nécessaire et pertinent.*

Les éléments à développer sont toujours écrits en **gras.**

## 1. Initialisation

Une Canvas est définie par sa largeur (width) et sa hauteur (height), en pixel, ainsi que par les pixels contenue à l'intérieur.

```
case class Canvas(
  width: Int = 0,
  height: Int = 0,
  pixels: Vector[Vector[Pixel]] = Vector()
)
```

Pour une canvas de taille `width = 3, height = 4` nous avons donc 4 lignes contenant chacune 3 pixels.

Chaque pixel est définie par :

- Sa position (x, y), avec `x ∈ [0, width - 1]` et `y ∈ [0, height - 1]`
- Sa *couleur*, définie ici comme étant un *character* (par exemple `'x', '.', '#'`, …)

*L'indexation commence à 0, c'est à dire que le pixel en haut à gauche a pour coordonnée `(x = 0, y = 0)`*

| | | |
|---|---|---|
| `Pixel(0, 0, '#')` | `Pixel(1, 0, '.')` | `Pixel(2, 0, '#')` |
| `Pixel(0, 1, '#')` | `Pixel(1, 1, '.')` | `Pixel(2, 1, '#')` |
| `Pixel(0, 2, '#')` | `Pixel(1, 2, '.')` | `Pixel(2, 2, '#')` |
| `Pixel(0, 3, '#')` | `Pixel(1, 3, '.')` | `Pixel(2, 3, '#')` |

### A. Dummy Canvas

Pour commencer à travailler, il existe actuellement trois actions déjà implémentées :

- `exit` - Quitte le programme
- `dummy` - Créer une canvas pré-définie de taille 3 x 4
- `dummy2` - Créer une canvas pré-définie de taille 3 x 1

Ainsi si vous entrez la commande dummy :

```
$ scala MainEmpty.scala
Starting application

======
Canvas:
Empty Canvas

Action: dummy

======
Canvas:
Size: 3 x 4
```

## B. Affichage

**Compléter les méthode `display` pour qu'elle affiche dans la console (*en utilisant print / println*) la canvas.**

Par exemple pour la canvas dummy, on s'attend à avoir :

```
Action: dummy

======
Current canvas:
Size: 3 x 4
#.#
#.#
#.#
#.#
```

## C. Pixel

On définit un pixel par sa position (`x: Int, y: Int`) et sa couleur (`color: Char`).
La case class Pixel existe déjà avec comme caractère par défaut un espace

```
case class Pixel(x: Int, y: Int, color: Char = ' ')
val p = Pixel(5, 3, 'o')
```

Cependant les actions vont recevoir comme argument des coordonnées sous forme de string "`x,y`".

**Implémenter les méthodes apply du companion object Pixel. On suppose que la string d'entrée s est *toujours* au format "x,y"**.

*Pensez à regarder du côté de la méthode `.split` des string.*

**RAPPEL COMPANION OBJECT**
*Rappel : Un `companion object` permet à la `class companion` d'accéder aux fonctions et membres privés de l'`object`. Il permet aussi entre autre de pouvoir surcharger le constructeur d'une class (cf. **TD 2 - Basiques Prog. Scala chapitre Object**)*
*Exemple :*

```
case class Person(fullname: String)

object Person {
  def apply(firstname: String, lastname: String): Person =
    Person(s"$firstname $lastname")
}

val p1 = Person("John", "DOE") // Person("John DOE")
```

On peut désormais instancier un Pixel de la façon suivante :

```
val p1 = Pixel("5,3", 'o')
val p2 = Pixel("2,4")
```

Pour vérifier votre code, vous pouvez lancez l'action `dummy2` et voir la canvas suivante :

```
#.#
```

## D. New Canvas

Pour pouvoir dessiner sur une canvas il est nécessaire d'être capable d'en créer une.

Vous allez implémenter la première action :

```
new_canvas [width] [height] [character]
```

Cette action s'attend à avoir 3 arguments :

- `width` : la largeur de la canvas
- `height` : la hauteur de la canvas
- `character` : le character par défaut de chaque pixel

Exemple

```
======
Current canvas:
Empty Canvas

Action: new_canvas 5 3 .

======
Current canvas:
Size: 5 x 3
.....
.....
.....
```

Pour exécuter une action, le programme a besoin d'une fonction avec la signature suivante :

```
def myAction(arguments: Seq[String], canvas: Canvas): (Canvas, Status)
```

- `arguments` : représente la liste des arguments donnés à l'action. Dans l'exemple ci-dessus ce serait : `Seq("5", "3", ".")`
- `canvas` : l'objet Canvas actuelle, au début c'est une Canvas vide

La fonction doit retourner un tuple contenant une Canvas (la même qu'en entrée avec des modifications, ou une nouvelle Canvas), ainsi qu'un Status. Dans un premier temps, renvoyez toujours un Status vide : `Status()`, nous traiterons les erreurs plus tard.

**Implémenter une méthode avec la même signature permettant de créer une Canvas en s'attendant à avoir trois arguments sous forme de String : `Seq(width, height, defaultChar)`**
**(Cette méthode peut appartenir à n'importe quel élément, nous vous suggérons de la faire appartenir à l'`object Canvas`) - Vous pouvez vous inspirer de la méthode `Canvas.dummy` ligne 153**

*Pour caster un string en entier il est possible d'utiliser ma méthode `.toInt`*

Pour que votre fonction soit reconnue par le programme, il faut la déclarer dans le fonction `execute` (ligne 41)

**Ajouter un *case* à la ligne 49 pour associer l'action `new_canvas` à la méthode que vous venez de créer. Exemple :**

```
case "my_new_cation" => Canvas.myAction
```

Normalement à ce stade là, vous pouvez lancer les actions `dummy` et `new_canvas [width] [height] [character]` :

```
Starting application

======
Current canvas:
Empty Canvas

Action: dummy

======
Current canvas:
Size: 3 x 4
#.#
#.#
#.#
#.#

Action: new_canvas 6 3 o

======
Current canvas:
Size: 6 x 3
oooooo
oooooo
oooooo

Action: exit

Exiting application
Final status Received exit signal
```

# 2. Actions

Théoriquement, toutes les actions décrites ci-dessous peuvent être implémentées dans n'importe quel ordre. Nous vous suggérons cependant l'ordre de suivant (difficulté croissante).
*Il est possible de faire la partie **3. Gestion des erreurs** même si la partie **2. Actions** n'est pas complète*

Chaque action nécessite de créer une méthode ainsi que de la déclarer au programme dans la méthode `execute` (comme à la question 1.C.)

## A. Load Image

L'action `load_image` a pour but de créer une canvas à partir d'un fichier existant (exemple le fichier `triforce` fourni avec l'énoncé)

```
load_image [filename]
```

Cette action prend un argument :

- `filename` - Nom du fichier à lire

La canvas créée doit contenir autant de ligne que le fichier. On suppose que toutes les lignes ont la même taille.


**Implémenter l'action load_image.**

Pour lire un fichier, importez le package `scala.io.Source`, vous pouvez ensuite récupérer le contenu d'un fichier sous forme de Vector de String (chaque élément représente une ligne du fichier) :

```
val content: Vector[String] = Source.fromFile("my_file").getLines().toVector
```

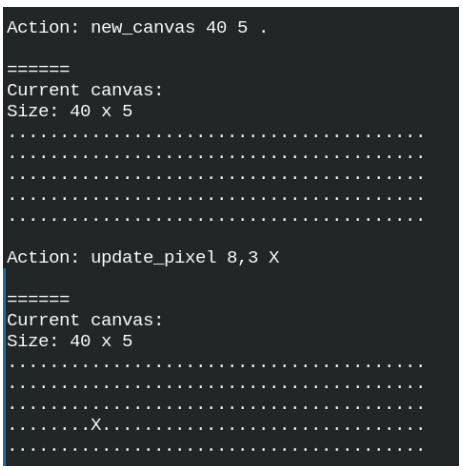
*Il convient ensuite de transformer ce `Vector[String]` en une `Canvas` (qui prend en argument un `Vector[Vector[Pixel]]`*

## B. Update Pixel

L'action `update_pixel` permet de mettre à jour un pixel à partir de ses coordonnées, en changeant sa couleur.

```
update_pixel [pixel] [color]
```

- `pixel` : Les coordonnées du pixel à modifier, au format "`x,y`"
- `color` : Le nouveau caractère (i.e. la couleur) du pixel



**Implémenter l'action `update_pixel`**
**Il est suggéré d'implémenter la méthode `update` de la `case class Canvas` qui renvoie une nouvelle Canvas contenant la modification.**

**Cette méthode prend un argument un `Pixel(x, y, color)` et remplace le pixel existant déjà dans le canvas pour les même coordonnées `(x, y)`**

**RAPPEL UPDATE**
*Les collections possèdent toujours une méthode updated permettant de renvoyer une nouvelle collection avec élément modifié*

```
val v = Vector("a", "b", "c")
val v2 = v.updated(1, "B")
println(v2) // Vector("a", "B", "c")
```

## C. Draw (line - v1)

Maintenant que l'on sait mettre à jour un pixel, nous allons pouvoir dessiner des figures avec l'action draw. L'action draw prend en premier argument le type de figure, puis les arguments nécessaires pour la dessiner.

Nous allons commencer par dessiner des lignes verticales et horizontales uniquement.

```
draw line [pixel1] [pixel2] [color]
```

- pixel1 : Les coordonnées du pixel de départ, au format "`x,y`"
- pixel2 : Les coordonnées du pixel d'arrivée, au format "`x,y`"
- color : La couleur de la ligne (i.e. de tous les pixels entre celui de départ et d'arrivée)

```
Action: new_canvas 40 5 .

======
Current canvas:
Size: 40 x 5
........................................
........................................
........................................
........................................
........................................

Action: draw line 4,3 14,3 x

======
Current canvas:
Size: 40 x 5
........................................
........................................
........................................
....xxxxxxxxxx..........................
........................................

Action: draw line 18,1 18,4 y

======
Current canvas:
Size: 40 x 5
........................................
.................y......................
.................y......................
....xxxxxxxxxx...y......................
.................y......................
```

Dans un premier temps, l'action ne s'occupent que du cas ou `pixel1.x == pixel2.x` OU `pixel1.y == pixel2.y` - c'est à dire une ligne verticale ou horizontale

**Implémenter l'action `draw line` pour les lignes horizontales et verticales**

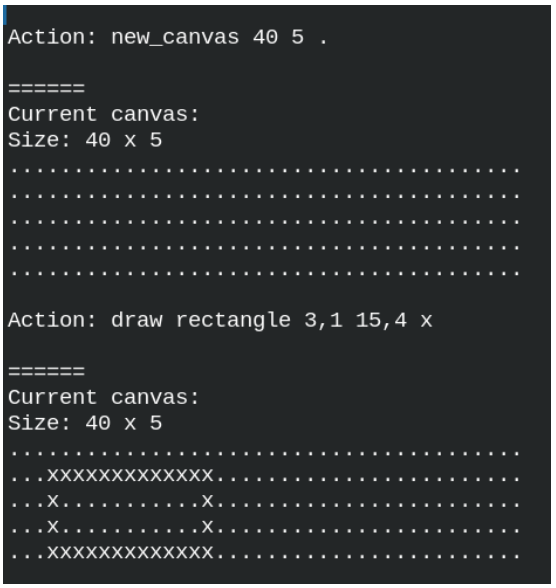
*La méthode `.updates` de la `case class Canvas` peut vous aider*

## D. Draw (rectangle)

À partir de lignes horizontales et verticales, il est trivial de dessiner un rectangle.
La deuxième figure à dessiner et donc un rectangle, à partir de deux angles opposés

```
draw rectangle [pixel1] [pixel2] [color]
```

- pixel1 : Les coordonnées d'un angle du rectangle, au format "`x,y`"
- pixel2 : Les coordonnées de l'angle opposé au format "`x,y`"
- color : La couleur de contour du rectangle




**Implémenter l'action `draw rectangle`, en utilisant le code déjà écrit pour dessiner une ligne.**

## E. Fill

On aimerait désormais être capable de colorier une forme de la même façon que le pot de peinture de *paint* : Lorsque l'on clique sur un pixel, l'algorithme va colorer ce pixel et tous les autres pixels de couleur identique.

Par exemple, prenons la forme suivante :

```
........o.........
...xxxxxoxxxxxxx...
...x....o......x...
...x....o......x...
...xxxxxoxxxxxxx...
```

Si on décide de remplir à partir du pixel `11,3` avec la couleur '*', alors tous les pixels adjacent de couleur '.' vous êtes remplacé par un '*' jusqu'à ne plus trouver de pixel '.'

1.


2.


3.


4.


5.


```
draw fill [pixel] [new_color]
```

- `pixel` : Pixel de départ à partir duquel colorier, au format “x,y”
- new_color : Nouvelle couleur à appliquer

**En utilisant de la récursivité, écrire l’algorithme de remplissage décrit ci-dessus**

## F. Draw (line - v2) - BONUS

*Les questions suivantes sont plus complexes et ne sont pas obligatoires, mais peuvent apporter quelques points bonus.*

On souhaite désormais être en mesure de dessiner tous types de ligne.
Pour cela nous allons utiliser l’algorithme de Bresenham.

Dans un premier temps, l’algorithme se place dans un cas particulier :

- La pente a un indice plus faible que 1. Autrement dit l’écart entre le début et la fin et plus important en *largeur* qu’en *hauteur* $\frac{y2-y1}{x2-x1} \leq 1 \Leftrightarrow y2 - y1 \leq x2 - x1$
- Le premier point est placé en haut à gauche du deuxième point ($x1 \leq x2$; $y1 \leq y2$)

La logique de l’algorithme est :

- On commence au point de départ $x, y$
- Puis à chaque étape
  - On se décale d’un pixel sur la droite $x = x + 1$
  - Pour cette nouvelle abscisse on regarde si la courbe est plus proche de $y$ ou de $y + 1$ et on modifie $y$ si nécessaire

L'algorithme est le suivant, avec D la variable permettant de savoir si le nouveau point est plus proche de $y$ ou de $y + 1$ , et plot la fonction permettant de dessiner le pixel

```
plotline(x0, y0, x1, y1)
    dx = x1 - x0
    dy = y1 - y0
    D = 2*dy - dx
    y = y0

    for x from x0 to x1
        plot(x,y)
        if D > 0
            y = y + 1
            D = D - 2*dx
        end if
        D = D + 2*dy
```

**Implémenter cet algorithme pour être capable de dessiner une ligne dans le cas particulier où le premier pixel est en haut à gauche du deuxième et où la pente est inférieure à 1.**

Exemple :

```
Action: new_canvas 40 5 .

======
Current canvas:
Size: 40 x 5
........................................
........................................
........................................
........................................
........................................

Action: draw line 1,1 20,3 x

======
Current canvas:
Size: 40 x 5
........................................
.xxxxx..................................
......xxxxxxxxxx........................
..............xxxxx.....................
........................................
```

## G. Draw (line - v3) - BONUS

On souhaite désormais être capable de gérer tous les cas.

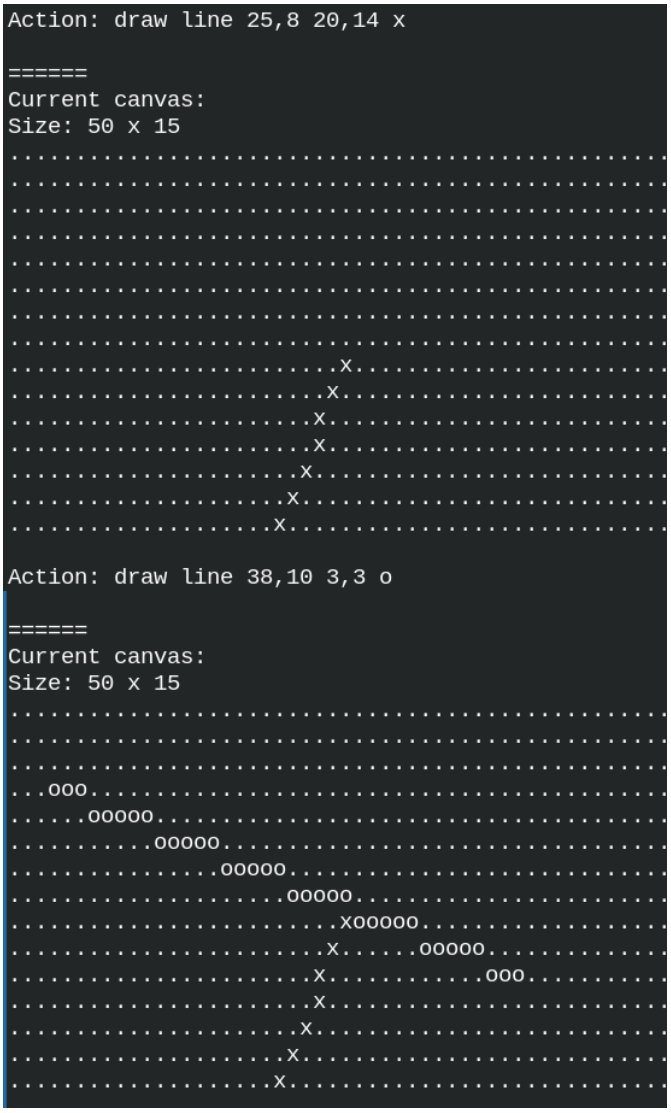
Tout d'abord, si on reste dans le cas $dx > dy$, il faut penser aux différentes situations :

- Si le deuxième point est en haut à droite du premier, la logique est exactement la même, à la seule différence que l'on fait $y = y - 1$ ou lieu de $y = y + 1$
- Si le premier point est à droite du deuxième point, il suffit d'inverser les deux points

Enfin, dans le cas où $dx < dy$, la logique est exactement la même, où incrémente $y$ de 1 à chaque étape, et on incrémente $x$ uniquement si nécessaire.
D est alors calculé de la même façon en inversant `dx` et `dy` dans l'algorithme.

**Modifier la fonction pour prendre en compte tous les cas.**

Vous êtes désormais capable de dessiner n'importe quelle ligne

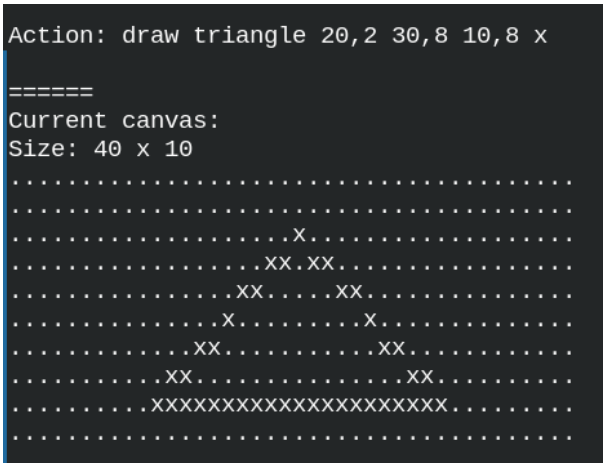



## H. Draw (triangle) - BONUS

Dessiner un triangle est maintenant trivial. Il suffit de dessiner trois segments pour relier les trois points.

```
draw triangle [pixel1] [pixel2] [pixel3] [color]
```

- `pixel1 / pixel2 / pixel3` : Trois sommets du triangle, au format “x,y”
- color : Couleur des segments




**Implémenter l'action `draw triangle`**

## I. Draw (polygon) - BONUS

Finalement, on peut aller jusqu'à dessiner n'importe quel polygone à partir d'une liste de points, en reliant les points 1 un 1.

```
draw polygon [pixel1] ... [pixeln] [color]
```

- `pixel1 / … / pixeln` : Ensemble de pixels à relier par des segments
- color : Couleur des segments

```
Action: draw polygon 1,1 30,2 20,8 15,6 3,9 x

======
Current canvas:
Size: 40 x 10
........................................
.xxxxxxxxxxxxxx.........................
.x.............xxxxxxxxxxxxxx..........
.x.........................xx..........
..x.......................xx............
..x......................x..............
..x...........xxx......xx................
..x.......xxxx...xx..xx..................
...x..xxxx.........xx....................
...xxx...................................
```

**Implémenter l'action `draw polygon`**

# 3. Gestion des erreurs

Pour que le programme soit plus accueillant pour des utilisateurs, on veut être capable de gérer les erreurs utilisateurs.
Le programme est capable d'afficher une erreur grâce à l'objet Status. Actuellement, on renvoie toujours un Status vide.

Si on renvoie un Status avec les attribut error et message, le programme l'affiche à l'utilisateur

Par exemple :

```
def default(arguments: Seq[String], canvas: Canvas): (Canvas, Status) =
 (canvas, Status(error = true, message = s"Unknown command"))
```

```
Action: hi
ERROR: Unknown command
```

**Ajouter de la gestion d'erreur pour renvoyer un message à l'utilisateur si :**

- **Le nombre d'arguments d'une action est faux**
- **Le format des arguments est mauvais**
- **L'action n'est pas possible, par exemple si un utilisateur veut mettre à jour un pixel en dehors de la canvas, ou créer une canvas avec des dimensions négatives, …**
- **N'importe quelle autre erreur venant de l'utilisateur**