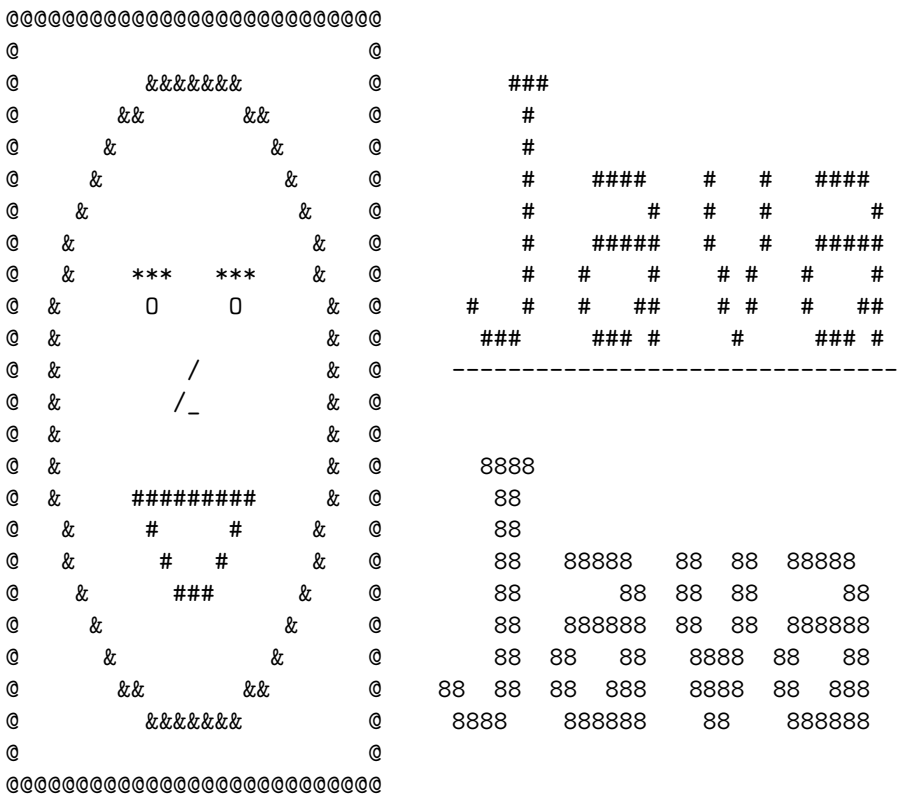
IFT1025 - Travail Pratique 1 (complet)

Concepts appliqués : Programmation orientée objet, algorithmique graphique 2D, héritage, développement d'application de taille moyenne

---

1 Contexte

Pour ce premier travail pratique, vous devrez concevoir un petit programme pour faire des dessins sur une ligne de commande, par exemple :



2 Spécification

Le programme s'exécute interactivement et doit répondre à certaines commandes. Un exemple d'exécution serait :

```
init 30 15
car .
ajouter rectangle 0 0 30 15
car #
ajouter rectangle 5 5 20 10
dessiner
```

qui va successivement :

1. Initialiser la surface de dessin à une surface de largeur 30 et de hauteur 15 (donc 15 lignes de 30 caractères)
2. Définir le caractère de dessin comme étant `.`
3. Ajouter un rectangle à la position $(0, 0)$ large de 30 caractères et haut de 15 caractères, qui sera dessiné avec des `.` (les coordonnées (x,y) sont données par rapport au coin supérieur gauche, qui a la position (0,0)).
4. Définir le caractère de dessin comme étant `#`
5. Ajouter un rectangle à la position $(5, 5)$ de taille 20 par 10 (dessiné avec des `#`)
6. Dessiner le tout sur la console, ce qui donnerait finalement :

```
..............................
..............................
..............................
..............................
..............................
.....##########................
.....##########................
.....##########................
.....##########................
.....##########................
.....##########................
.....##########................
.....##########................
.....##########................
.....##########................
```

2.1 Commandes

2.2 `init LARGEUR HAUTEUR`

Va initialiser la surface de dessin avec une surface vide de taille `LARGEUR` par `HAUTEUR`, par exemple :

```
init 70 25
```

définit que la surface de dessin sera de 25 lignes de 70 caractères chacune.

2.3 `car CARACTERE`

Va définir le caractère de dessin actuel à `CARACTERE`, par exemple :

```
car #
```

Définit que le caractère utilisé sera `#` pour les prochaines formes ajoutées.

2.4 `ajouter FORME ...`

Va ajouter une nouvelle forme à la surface de dessin. Les paramètres dépendent de la forme à ajouter (voir plus bas pour les différentes formes disponibles).

On pourrait par exemple ajouter un rectangle au dessin avec la commande :

```
ajouter rectangle 5 5 20 10
```

La forme ajoutée sera dessinée avec le caractère de dessin actuel. Si on faisait quelque chose comme :

```
car #
ajouter rectangle 5 5 20 10
car 8
ajouter rectangle 2 1 5 5
ajouter rectangle 4 2 8 7
```

Ça ajouterait d'abord un rectangle qui serait dessiné avec des `#`, puis ça ajouterait deux rectangles dessinés avec des `8`.

S'il n'est pas spécifié, le caractère de dessin est initialement un dièse (`#`).

2.5 `dessiner`

Va simplement afficher sur la console l'ensemble des formes ajoutées. Les formes sont dessinées dans l'ordre dans lequel elles sont ajoutées.

Par exemple la suite de commandes :

```
init 10 10
car .
ajouter rectangle 0 0 10 10
car #
ajouter rectangle 3 3 4 4
car 8
ajouter rectangle 1 1 4 4
dessiner
```

Affichera sur la console :

```
..........
.8888.....
.8888.....
.8888##...
.8888##...
...####...
...####...
..........
..........
..........
```

Ici, le troisième rectangle ajouté s'affiche "devant" les autres.

2.6 `brasser`

Va déplacer toutes les formes de ± 1 caractère sur l'axe des X, des Y ou les deux, où le déplacement positif ou négatif et les axes sont choisis de manière aléatoire et différemment pour chaque forme du dessin.

Par exemple la suite de commandes :

```
init 10 10
car .
ajouter rectangle 0 0 10 10
car #
ajouter rectangle 3 3 4 4
car 8
ajouter rectangle 1 1 4 4
brasser
dessiner
```

Pourrait donner :

```
8888
8888......
8888......
8888......
..####....
..####....
..####....
..####....
..........
..........
```

Dans cet exemple, le premier rectangle (`.`) a été déplacé de $(0, +1)$ par rapport à sa position originale, le deuxième (`#`) a été déplacé de $(-1, 0)$ et le troisième (`8`) a été déplacé de $(-1, -1)$.

Notez que même si une forme peut avoir des coordonnées qui dépassent les bornes de la zone de dessin, on n'affiche toujours qu'une surface de la taille de la surface de dessin.

Dans le cas où il n'y a aucune forme en un point (par exemple, à droite du rectangle sur la première ligne), on affiche un espace.

2.7 `renverser`

Renverse verticalement le dessin.

Par exemple :

```
init 10 10
car .
ajouter rectangle 0 0 10 10
car #
ajouter rectangle 3 3 4 4
car 8
ajouter rectangle 1 1 4 4
renverser
dessiner
```

Donnerait le résultat :

```
..........
..........
..........
...####...
...####...
.8888##...
.8888##...
.8888.....
.8888.....
..........
```

Notez qu'utiliser `renverser` deux fois de suite est équivalent à ne rien faire.

2.8 Formes dessinables

Plusieurs types de formes peuvent être ajoutées au dessin. On vous rappelle que les coordonnées (x, y) sont données par rapport au coin supérieur gauche, qui a la position $(0, 0)$.

2.8.1 `ajouter rectangle X Y LARGEUR HAUTEUR`

Ajoute un rectangle de taille `LARGEUR` par `HAUTEUR` dont le coin supérieur gauche est sur le point (X, Y).

Par exemple :

```
init 10 10
car .
ajouter rectangle 0 0 10 10
dessiner
```

Donne le résultat :

```
..........
..........
..........
..........
..........
..........
..........
..........
..........
..........
```

2.8.2 `ajouter carre X Y COTE`

Ajoute un carré dont le coin supérieur gauche est à la position (X, Y) dont la taille d'un côté est de `COTE` caractères.

La commande `ajouter carre 0 0 10` aura donc le même effet que `ajouter rectangle 0 0 10 10`.

2.8.3 `ajouter ligne X1 Y1 X2 Y2`

Ajoute une ligne allant du point $(X1, Y1)$ inclusivement jusqu'au point $(X2, Y2)$ inclusivement.

Par exemple :

```
init 10 10
car .
ajouter ligne 0 0 9 9
dessiner
```

Donne le résultat :

```
#.........
.#........
..#.......
...#......
....#.....
.....#....
......#...
.......#..
........#.
.........#
```

2.8.4 `ajouter cercle X Y RAYON`

Ajoute un cercle de rayon `RAYON` dont le centre est à la position (X, Y).

Par exemple :

```
init 10 10
car .
ajouter rectangle 0 0 10 10
car #
ajouter Cercle 4 4 3
dessiner
```

Donne le résultat :

```
..........
...###....
..#...#...
.#.....#..
.#.....#..
.#.....#..
..#...#...
...###....
..........
..........
```

2.8.5 `ajouter caractere X Y CARACTERE`

Ajoute le caractère `CARACTERE` sur la surface à la position (X, Y).

Par exemple :

```
init 8 11
car .
ajouter rectangle 0 0 8 11
car #
ajouter lettre 0 0 B
dessiner
```

Donne le résultat :

```
........
.#####..
..#...#.
..#...#.
..#...#.
..####..
..#...#.
..#...#.
..#...#.
.#####..
........
```

Note: Les représentations des caractères possibles vous seront fournies plus tard. Vous n'aurez pas à coder chaque lettre à la main.

2.8.6 `ajouter texte X Y TEXTE`

Ajoute la séquence de caractères `TEXTE` sur la surface à la position (X, Y).

Par exemple :

```
init 30 11
car #
ajouter texte 0 0 Java
dessiner
```

Donne le résultat :

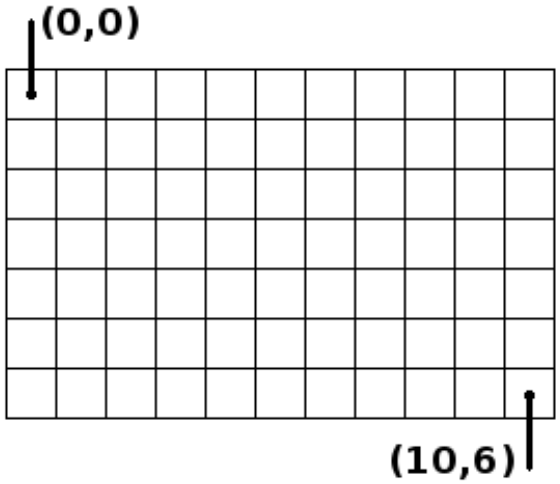
```
   ###
    #
    #
    #    ####    #   #   ####
    #       #    #   #       #
    #    #####   #   #   #####
    #   #    #    # #   #    #
#   #   #   ##    # #   #   ##
 ###     ### #     #     ### #
```

3 Précisions supplémentaires

3.1 Coordonnées

Les coordonnées sont calculées à partir du coin supérieur gauche et commencent à zéro.

Par exemple, une grille de taille 7 par 11 aura les coordonnées (x, y) suivantes aux extrémités :



3.2 Caractère par défaut

Si aucun caractère n'est dessiné dans une des cases, le caractère d'arrière plan par défaut est l'espace. Autrement dit, l'appel :

```
init 5 5
dessiner
```

Devrait afficher 5 lignes contenant seulement 5 espaces chacune.

3.3 Classe principale

La classe principale de votre programme doit se nommer `Tp1`.

On devrait donc pouvoir compiler et exécuter votre programme en tapant en ligne de commande :

```
javac Tp1.java
java Tp1
```

3.4 Scanner

Pour lire des instructions sur la ligne de commande, utilisez un `Scanner`. Le programme se termine lorsqu'on entre "fin" ou lorsqu'il n'y a plus rien à lire sur la ligne de commande (sous Unix/Linux/Mac, lorsqu'on entre *Ctrl+D*).

Vous pouvez vous baser sur le code suivant :

```
Scanner scan = new Scanner(System.in);

String instruction;

do {
    instruction = scan.nextLine();

    if(instruction.equals("fin")) {
        break;
    }

    // Faire quelque chose avec l'instruction
    // ...

} while (scan.hasNext());
```

3.5 `init`

Notez qu'on devrait pouvoir afficher plusieurs dessins lors d'une même exécution du programme. Cela signifie que la commande `init` devrait pouvoir être appelée plus d'une seule fois et devra réinitialiser la surface de dessin à chaque fois.

Si on tente de dessiner des formes avant d'avoir fait un premier `init`, on devrait afficher le message d'erreur suivant sur la console :

```
ERREUR: Aucune surface définie
```

Et continuer l'exécution du programme.

4 Algorithmes et Fichiers fournis

4.1 Dessiner du texte

Pour dessiner du texte, vous pouvez utiliser la classe `Police8x12.java` fournie sur StudiUM. Référez-vous au code source pour plus d'informations sur son utilisation.

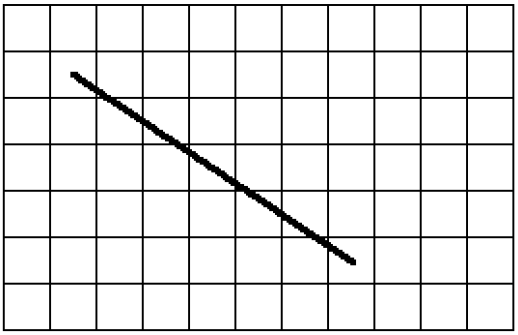
Vous aurez besoin des opérateurs bit-à-bit `&` et `>>` pour décoder les `bytes`. Référez-vous à la documentation ou aux notes de cours de Programmation 1 au besoin.

4.2 Dessiner une ligne

Pour dessiner une ligne discrétisée correctement, on utilisera un algorithme inspiré de l'*algorithme de tracé de segment de Bresenham*.

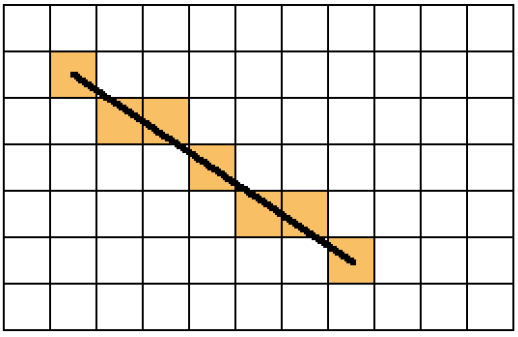
L'algorithme a pour but de trouver quelles cases sur une grille doivent être dessinées pour donner la ligne discrète la plus proche possible d'une ligne idéale (fractionnaire) allant du point $p_1 = (x_1, y_1)$ jusqu'au point $p_2 = (x_2, y_2)$.

Donnons d'abord un exemple : tracer une ligne entre les points $(1, 1)$ et $(7, 5)$.



Dans cet exemple, la ligne "avance" plus vite horizontalement que verticalement, puisque $x_2 - x_1 > y_2 - y_1$. Il serait donc logique de tracer la ligne avec une seule case par colonne x, mais possiblement avec plus d'un seul point par rangée y.

L'idée sera donc de parcourir une à une chaque colonne x allant de x_1 à x_2, en choisissant pour chaque valeur de x la colonne y la plus proche de la ligne en ce point, ce qui donnerait ici :



L'équation du segment de droite étant :

$$\frac{y - y_1}{y_2 - y_1} = \frac{x - x_1}{x_2 - x_1}$$

On peut isoler y et trouver que le y le plus proche de la ligne en un point x connu est :

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1$$

Pour notre exemple, le point de départ est $x = 1$, ce qui donne $y = 1$. Le point suivant, $x = 2$, aurait une valeur en y de 1.666667. Le pixel le plus proche peut être obtenu en arrondissant cette valeur (avec `Math.round()`), ce qui nous donne le pixel $(2, 2)$. On peut continuer avec $x = 3, 4, 5, 6, 7$.

Dans le cas où on aurait plutôt $\Delta y > \Delta x$, la logique s'applique toujours, mais on aurait une seule case par rangée y au lieu d'une seule case par colonne. Il faudrait donc plutôt parcourir les rangées y allant de y_1 à y_2, en trouvant la colonne x la plus proche de la ligne pour chaque y donné.

En isolant x, on obtient une formule très similaire à la précédente :

$$x = \frac{x_2 - x_1}{y_2 - y_1}(y - y_1) + x_1$$

Bien que des versions optimisées de cet algorithme soient disponibles sur internet, on vous demande d'implanter la version décrite ici pour ce TP.

Assurez-vous que votre code fonctionne pour tous les types de lignes. Voici un exemple complet pour vous aider à vérifier votre fonction :

Code:

```
init 31 31
car .
ajouter rectangle 0 0 31 31
car 1
ajouter ligne 15 15 30 20
car 2
ajouter ligne 15 15 30 10
car 3
ajouter ligne 15 15 0 20
car 4
ajouter ligne 15 15 0 10
car 5
ajouter ligne 15 15 20 30
car 6
ajouter ligne 15 15 10 30
car 7
ajouter ligne 15 15 20 0
car 8
ajouter ligne 15 15 10 0
car h
ajouter ligne 0 15 30 15
car v
ajouter ligne 15 0 15 30
```

Résultat:

```
..........8....v....7..........
..........8....v....7..........
...........8...v...7...........
...........8...v...7...........
...........8...v...7...........
............8..v..7............
............8..v..7............
............8..v..7............
.............8.v.7.............
.............8.v.7.............
44..........8.v.7..........22
..444.........8v7.........222..
.....444......8v7......222.....
........444...8v7...222........
...........444.v.222...........
hhhhhhhhhhhhhhhvhhhhhhhhhhhhhhh
...........333.v.111...........
........333...6v5...111........
.....333......6v5......111.....
..333.........6v5.........111..
33...........6.v.5...........11
.............6.v.5.............
.............6.v.5.............
............6..v..5............
............6..v..5............
............6..v..5............
...........6...v...5...........
...........6...v...5...........
...........6...v...5...........
..........6....v....5..........
..........6....v....5..........
```

4.3 Dessiner un cercle

L'algorithme de Bresenham pour tracer des cercles complets est donné en pseudo-code sur Wikipédia.

```
procédure tracerCercle (entier rayon, entier x_centre, entier y_centre)
    déclarer entier x, y, m ;
    x <- 0
    y <- rayon        // on se place en haut du cercle
    m <- 5 - 4*rayon  // initialisation
    Tant que x <= y   // tant qu'on est dans le second octant
        tracerPixel( x+x_centre, y+y_centre )
        tracerPixel( y+x_centre, x+y_centre )
        tracerPixel( -x+x_centre, y+y_centre )
        tracerPixel( -y+x_centre, x+y_centre )
        tracerPixel( x+x_centre, -y+y_centre )
        tracerPixel( y+x_centre, -x+y_centre )
        tracerPixel( -x+x_centre, -y+y_centre )
        tracerPixel( -y+x_centre, -x+y_centre )
        si m > 0 alors
            y <- y - 2
            m <- m - 8*y
        fin si
        x <- x + 1
        m <- m + 8*x + 4
    fin tant que
fin de procédure
```

5 Tests de votre programme

Pour faciliter les tests de votre programme, écrivez-vous des fichiers qui contiennent du code pour tester. Par exemple, si le fichier `testRectangle.txt` contient :

```
init 10 10
car #
ajouter rectangle 0 0 10 10
dessiner
```

Vous pourrez tester votre programme avec ce code en lançant votre programme en ligne de commande avec :

```
java Tp1 < testRectangle.txt
```

Ce qui aura pour effet de prendre le contenu de `testRectangle.txt` comme si vous l'aviez tapé dans la console au moment d'exécuter le programme.

Testez **toutes** les fonctionnalités de votre programme, vous serez en partie évalués sur la qualité de vos tests.

6 Rapport

Remettez un fichier nommé `README.txt` dans lequel vous écrirez :

1. Les noms + UNIP (le numéro avec lequel vous vous logguez sur StudiUM – mais pas votre mot de passe) des coéquipiers qui ont travaillé sur le TP
2. Une explication de ce qui ne fonctionnait pas dans votre design initial et que vous avez dû changer au moment de coder le TP

Un fichier `README.txt` pourrait avoir l'air de :

```
-----------------------------------------------

Jimmy Whooper, p1234567
Xavier Whooper, p7654321


- Classe Blub, méthode foo(String x)

Cette méthode avait besoin d'avoir accès à la
variable Truc.bar, nous avons dû lui ajouter un
paramètre

- La classe Blob héritait de Blub

L'héritage n'était pas un bon choix ici, puisque
certains attributs de la classe Blub ne sont pas
reliés à la classe Blob

- Classe Abc, attribut defgh

Il manquait l'attribut defgh aux instances de la
classe Abc, ce qui était absolument nécessaire
pour noter l'état de l'alphabet


etc

-----------------------------------------------
```

Il serait surprenant que vous ayez respecté à la lettre chaque détail de votre design initial, parlez de ce qui manquait, de ce qui ne fonctionnait pas ou de quoi que ce soit d'autre qui est pertinent de mentionner.

7 Évaluation

Le barème pour ce travail est le suivant :

- 3/15 : Rapport
- 5/15 : Respect de la spécification
- 5/15 : Élégance et qualité du code
- 2/15 : Tests du programme

Remettez tous les fichiers dans une archive `.zip` (**pas** de `.rar` autorisés) qui contient la structure suivante :

```
tp1.zip:
|
|--- README.txt
|
|--- code/
|   |
|   |--- Tp1.java  (classe principale pour lancer votre programme)
|   |--- ...       (vos autres classes)
|
|--- tests/
    |
    |--- testRectangle.txt
    |--- testLigne1.txt
    |--- testLigne2.txt
    |--- (Mettez tous les tests que vous avez écrits, ils seront évalués)
```

- La date de remise est le **8 mars 2019 à 23h59**. Un travail remis le lendemain avant 23h59 aura une pénalité de *-5* points. Un travail remis passé le lendemain se verra attribué la note de zéro.
- Vous **devez** faire le travail par groupes de 2 personnes. Indiquez vos noms clairement dans les commentaires au début de votre code, dans le fichier principal, que vous nommerez `Tp1.java`.
- Une seule personne par équipe doit remettre le travail sur StudiUM
- Un travail fait seul engendrera une pénalité. Les équipes de plus de deux ne sont pas acceptées.
- De plus :
  - La performance de votre code doit être raisonnable
  - Chaque fonction devrait être documentée en suivant le standard `JavaDoc`
  - Il devrait y avoir des lignes blanches pour que le code ne soit pas trop dense (utilisez votre bon sens pour arriver à un code facile à lire)
  - Les identificateurs doivent être bien choisis pour être compréhensibles (évitez les noms à une lettre, à l'exception de i, j, ... pour les variables d'itérations des boucles for)
  - Vous devez respecter le standard de code pour ce projet, soit les noms de variables et de méthodes en camelCase