Gestion du mode de jeu :

Notre application doit laisser aux joueurs le choix d'une difficulté entre Normal et Avancée, l'unique différence entre ces deux modes de jeu est que 3 cases de la grille des scores donne une quantité minime de point.
On est tenté de déceler un schéma récurant dans la continuité de l'attribution des points, mais c'est très fastidieux, et il est difficile de former le tableau avec une boucle.
On a donc pensé qu'un moyen simple de définir la grille des scores est de d'abord la déclarer sous forme de tableau en mode Normal avec un bloc initialisateur, demander une saisie, et si l'utilisateur choisit « Avancée », on modifie les 3 cases qui changent de ce tableau.

Pour accéder au score d'une suite de longueur n, on consulte l'indice n – 1 du tableau.

```
// Bloc Initialisateur du tableau
tableScore = new int[] { 0, 1, 3, 5, 7, 9, 11, 15, 20, 25, 30, 35, 40, 50, 60, 70, 85, 100, 150, 300 } ;

// Demande une saisie tant qu'elle est incorrecte
do {

    Console.couleurFont(CouleurConsole.VERT) ;
    Console.print ( espaceAffichageInfo + "[========================================]\n" ) ;
    Console.couleurFont(CouleurConsole.VERT) ;
    Console.print ( espaceAffichageInfo + "     Mode ( Normal / Avancé ) : " ) ;

    difficulte = Clavier.lireString () ;

} while ( ! difficulte.equals ( "Normal" ) && ! difficulte.equals ( "Avancé" ) ) ;

// Si la difficulté est Avancée, change les points à certains indices du tableau
if ( difficulte.equals ( "Avancé" ) )
{
    tableScore[5] = 3 ;
    tableScore[11] = 20 ;
    tableScore[16] = 50 ;
}
```

Gestion de la pioche :

Les choix d'implémentations sont similaires à ceux du tableau des scores : un bloc initialisateur, une boucle.
On a d'abord créé la pioche au travers d'une boucle avec une logique d'itération simple.
Une boucle qui itère 30 fois pour les jetons allant de 1 à 30, une autre pour les doublons de 11 à 19 et enfin le joker qu'on stocke -1 (on ne peut stocker que des variables du même type dans un tableau).

```
final int TAILLE = 40 ;

/* Variables     */
/* - - - - - - - */

int[] tab ;

// Instanciation du tableau
tab = new int[TAILLE] ;

for ( int cpt = 0 ; cpt < 30 ; cpt++ )
{
    tab[cpt] = cpt + 1 ;
}

for ( int cpt = 30 ; cpt < 39 ; cpt++ )
{
    tab[cpt] = cpt - 19 ;
}

tab[39] = -1 ;
```

Mais on a préféré l'autre méthode du bloc, car il est plus simple de modifier le contenu de la pioche directement dans le bloc plutôt que changer la logique des boucles for.

Il nous faut ensuite mélanger notre pioche, rien de compliqué. On choisit au hasard deux indices du tableau de notre pioche et on échange les valeurs qu'ils référencent entre elles, et ce 1000 fois.

Pour « piocher », on choisit un indice aléatoire entre 0 et le nombre d'éléments de la pioche, on stocke la valeur qu'il référence dans une variable écaille, et on remplace cette valeur par le dernier élément de la pioche. On décrémente le nombre d'éléments de 1 pour que la pioche « oublie » les valeurs clonées.

Si jamais le joker est pioché, on stocke cet événement dans un booléen.

Petite particularité de notre pioche cependant, pour garantir l'homogénéité dans notre code. On a choisit de gérer notre pioche au travers d'un objet Sac.
Cet objet suit le diagramme UML suivant :

Sac	
- jetons - nbJeton - jokerEstPioche	: Tableau d'entiers : entier : booléen
+ Sac() + piocher() + getJokerEstPioche	 : entier : booléen

Précision pour .piocher() :

Si la pioche est vide, la méthode retourne 0 (même si cela n'est jamais sensé arriver)

Calcul du score à partir d'un serpent déjà rempli :

Tout tourne autour de la gestion de rupture.
On compare chaque valeur du serpent avec la valeur précédente et tester si elles forment une suite, et ce ainsi de suite.

Imaginons une suite :
01 12 14 16 21 09 12 15 28 02 14

Avant le début de notre boucle qui va parcourir l'ensemble de notre tableau. On va définir comme premier terme précédent la valeur à l'indice 0 du tableau (ici 01). On parcourt ensuite notre tableau à partir de l'indice 1 (ici 12). On test si le terme est supérieur ou égal au précédent.

A chaque terme qui rallonge la suite, on incrémente un compteur pour stocker le nombre de point que rapportera cette suite.

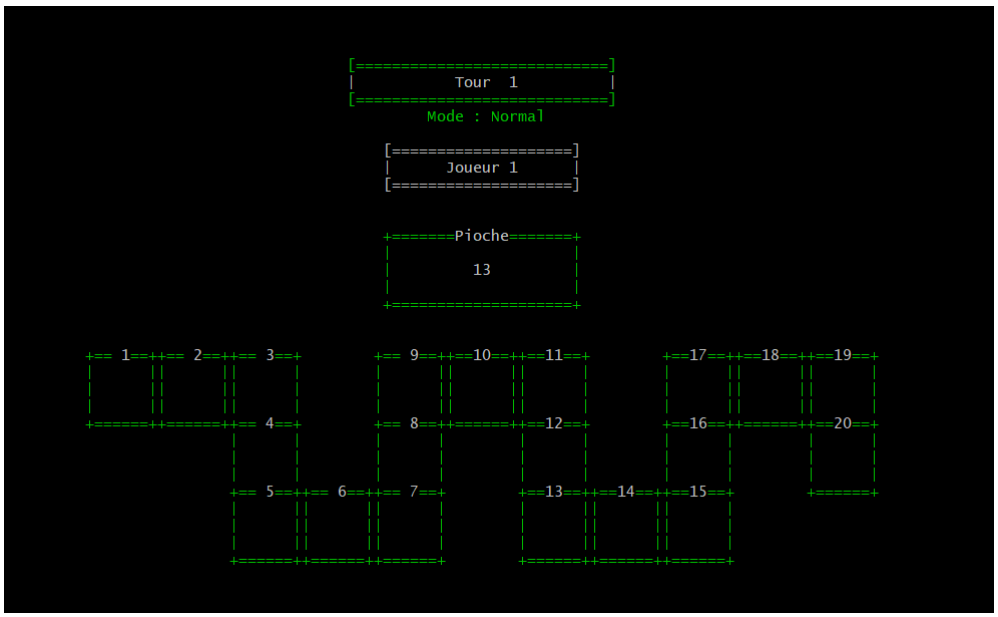
On concatène chaque terme qui forme une suite dans une String. Et si rupture il y a (ici 09), la String est affichée avec ses points et on repart sur une nouvelle chaîne, un nouveau compteur, et on incrémente la valeur en points de la suite à la somme des points.

A la sortie de la boucle, on affiche la dernière suite et le total des points.

```
suite =  "" ;

suiteNombre = 1 ;
sommeSuite = cptSuite = totalScore = 0 ;

termePrec = serpent[0] ;
suite = Integer.toString ( serpent[0] ) + " " ;
for ( int cpt = 1 ; cpt < serpent.length ; cpt++ )
{
    if ( termePrec <= serpent[cpt] )
    {
        suiteNombre++ ;
        suite = suite + Integer.toString ( serpent[cpt] ) + " " ;
    }
    else
    {
        Console.print ( String.format ( "%-60s", suite ) + " : " + score[suiteNombre - 1] + "\n" ) ;
        totalScore = totalScore + score[suiteNombre - 1] ;

        suiteNombre = 1 ;
        suite = Integer.toString ( serpent[cpt] ) + " " ;
    }
    termePrec = serpent[cpt] ;
}

Console.print ( String.format ( "%-60s", suite ) + " : " + score[suiteNombre - 1] + "\n" ) ;
totalScore = totalScore + score[suiteNombre - 1] ;

Console.print ( String.format ( "%-60s", "Total" ) + " : " + totalScore + "\n" ) ;
```

Élaboration de l’interface (GUI) :

On se fait un inventaire de ce que doit afficher notre interface :
- Le serpent du Joueur
- Le tour
- La pioche
- Le mode de jeu



Tous ces éléments ont leur affichage demandé au travers de méthode. Rien de spécial mise à part l’utilisation de la classe Console.
SAUF le Serpent qui lui est affiché grâce à un algorithme assez long.

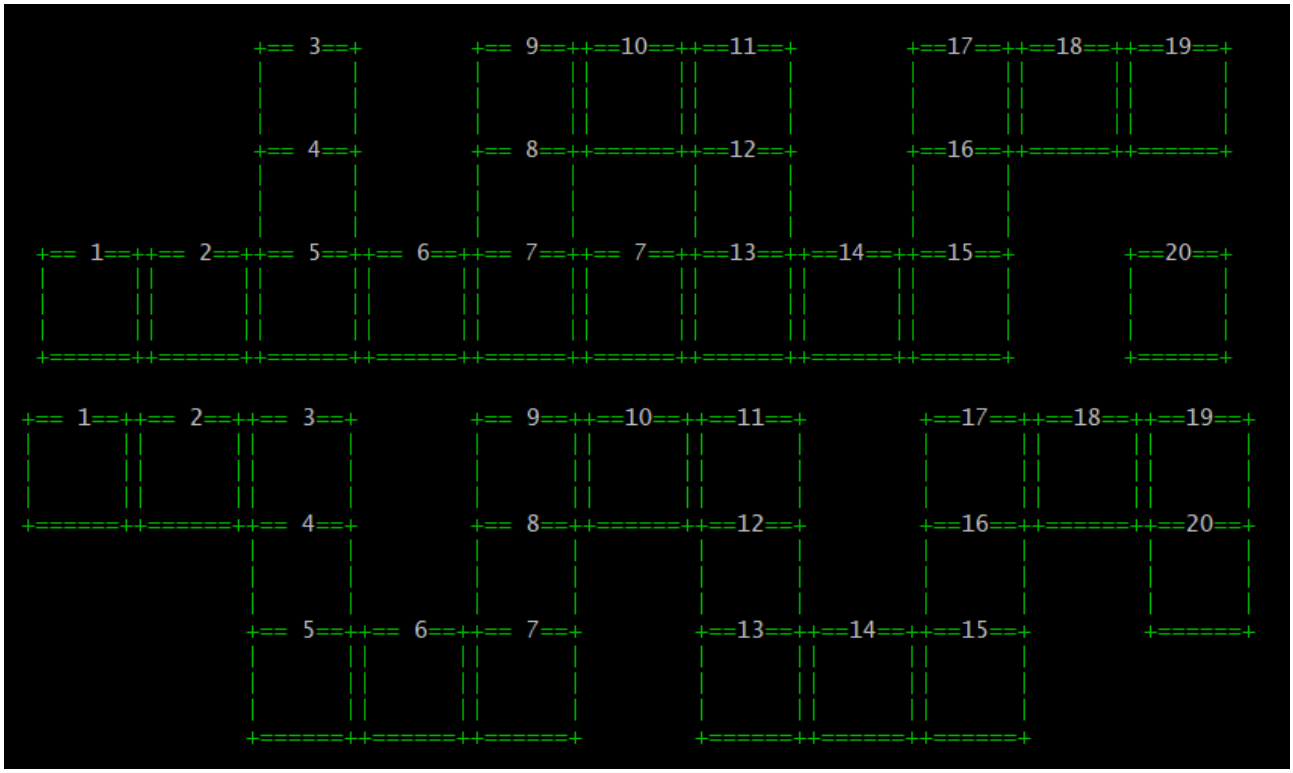
On aurait pu se contenter d’un affichage en brute avec des Strings interminables dans le code. Mais se n’est pas intellectuellement stimulant.

On a remarqué que les cases d’un serpent pouvaient être représentés sous forme d’une grille. Et on a voulu faire en sorte que la disposition des ces cases soit facilement modifiable si on en éprouvait le besoin.
Cette disposition, dans le code, est représentée par un tableau constant d’entier.

```
final int[][] DISPOSITION_SERPENT = {
    {  0,  1, 2, -1, 8,  9, 10, -1, 16, 17, 18 },
    { -1, -1, 3, -1, 7, -1, 11, -1, 15, -1, 19 },
    { -1, -1, 4,  5, 6, -1, 12, 13, 14, -1, -1 }
} ;
```

Les -1 représentent des blancs. Le reste, l’indice du tableau du serpent que la case doit afficher.
On parcourt cette disposition avec 2 boucles for imbriquées (et d’autres boucles, des conditions pour les cas particuliers, etc...)

```
final int[][] DISPOSITION_SERPENT = {
    { -1, -1, 2, -1, 8,  9, 10, -1, 16, 17, 18 },
    { -1, -1, 3, -1, 7, -1, 11, -1, 15, -1, -1 },
    {  0, 1,  4,  5, 6,  6, 12, 13, 14, -1, 19 }
} ;
```



Finalisation de l’application :

Le plus gros est déjà fait. Il ne reste qu’à mettre en forme le code et séparer chaque fonction.

Tout ce qui concerne la saisie et l’affichage se retrouve dans la classe Affichage. Le programme principale se contente de décomposer chaque tour en plusieurs étapes.

On fait appel à un Objet Sac qu’on a détaillé plus haut qui se trouve dans la classe Sac.

La liste des méthodes de la classe Affichage :
+ affichageSerpent(int[] serpent)
+ suiteSerpent(int[] serpent, int[] score, int joueur)
+ retourLigne(int nbRetourLigne)
+ saisieCase(int caseMax)
+ tourJoueur(int joeur, int[] serpent, int tour, int[] score, int tirage)
+ remplaceJoker(int[] serpent, int joueur) : int[] serpent
+ ecranVictoire(String message)