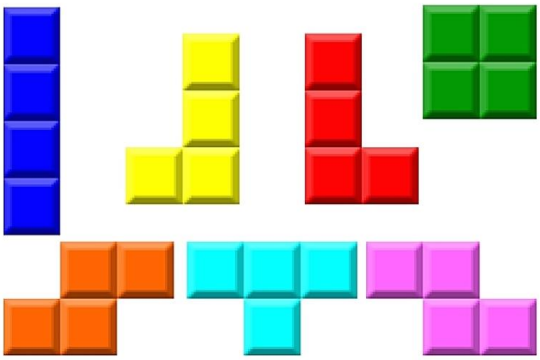
Mise en contexte ludique

Vous êtes briqueteur-maçons débutants tentant de percer dans le monde sans pitié de la maçonnerie et, de façon un peu naïve, décidez de prendre un contrat dans lequel vous devez construire un mur avec des briques de formes très particulières. Aussi, comme si le contrat n'était pas déjà assez ardu, le propriétaire du mur vous mentionne les détails suivants:

A. C'est lui qui vous fournira les briques une par une. L'ordre dans lequel il les donne est aléatoire.
B. Vous connaîtrez uniquement la brique que vous avez dans vos mains mais n'aurez aucune idée des briques futures.

Pris de panique, vous pensez démissionner à l'instant mais vous vous rendez compte d'un détail particulier concernant les briques : elles sont toutes d'une ou l'autre des 7 formes suivantes :



Étant très habile en programmation, vous décidez, avant de commencer le contrat, de programmer une solution logicielle qui vous aidera à mener ce contrat à terme.

Vous en êtes donc au jour un, assis devant votre ordinateur et prêt à commencer à programmer votre outil d'aide à l'empilement optimal de briques que vous avez scientifiquement surnommé le « SMEOBFP » (« Super machine d'empilement optimal de brique de formes particulières »).

Vous vous donnez donc le objectif d'implémentation suivant : Implémenter un algorithme qui permet de maximiser le nombre de lignes de briques complètes.

Spoiler alert! – **Vous devez implémenter une algo qui joue au Tetris!**

Solution Visual Studio fournie

Une solution Visual Studio comprenant un jeu fonctionnel et programmable de Tetris vous est fournie pour compléter le défi. Pour pouvoir programmer votre algorithme et l'injecter dans l'application, vous devez utiliser la fonction suivante :

```
public interface ITetrisAPI
  {
    /// <summary>
    /// Run the application with an action to execute every game tick
    /// </summary>
    /// <param name="_onTick">The action to run every game tick</param>
    void Run( Action<IState, ITetrisControllerAPI> _onTick );
  }
```

Concrètement, une méthode avec la signature suivante sera appelée à tous les « Tick » de la partie.

```
private static void OnTick( IState _state, ITetrisControllerAPI _controller )
{
  // code here...
}
```

Informations concernant le Tick

- Un tick correspond à un cycle de calcul du jeu. Lorsque vous jouez au vrai jeu de Tetris, le tick correspond à une unité temps écoulée, au cours de laquelle chaque le bloc courant va descendre automatiquement d'une case.
- À la fin d'un du tick, l'application détermine si le jeu doit continuer ou si la partie est terminée.
- L'algorithme est limité à 5 commandes par tick du jeu. Toute commande excédentaire ne sera pas communiquée au jeu.
- Les mouvements que vous programmez sont appliqués après la fin du tick (et non pendant). En d'autres termes, si vous faites des mouvements, _state.CurrentPiece n'est pas affectée par vos mouvements pendant l'exécution de la méthone OnTick; ils sont appliqués que lorsque la méthode a retourné.

IState

L'état du jeu (cases déjà remplies dans la grille et position de la pièce courante) est donnée par l'objet _state. Le détail de l'interface IState est présentée en Annexe 3.

Note : La pièce courante n'apparait pas dans l'objet « Grid ». L'objet « Grid » indique seulement les blocs qui ont été déposés à leur position finale dans des ticks précédents.

ITetrisControllerAPI

Pour bouger la pièce courante, l'interface ITetrisControllerAPI est la suivante :

```
public interface ITetrisControllerAPI
  {
    /// <summary>
    /// Returns the number of move that the controller can make during this tick
    /// </summary>
    /// <returns></returns>
    Int32 RemainingMoves { get; }

    /// <summary>
    /// Moves the current piece from one position to the left
    /// </summary>
    /// <returns>Whether the movement was successful or not</returns>
    Boolean TryMoveLeft();

    /// <summary>
    /// Moves the current piece from one position to the right
    /// </summary>
    /// <returns>Whether the movement was successful or not</returns>
    Boolean TryMoveRight();

    /// <summary>
    /// Moves the current piece from one position down
    /// </summary>
    /// <returns>Whether the movement was successful or not</returns>
    Boolean TryMoveDown();

    /// <summary>
    /// Rotates the current piece clockwise for 90 degrees
    /// </summary>
    /// <returns>Whether the movement was successful or not</returns>
    Boolean TryRotateShape();
  }
```

Configuration des parties

La génération de la suite de blocs présentés au joueur est semi-aléatoire et est généré à l'aide d'un *seed.* Ce *seed* est une chaine de caractère qui a pour objectif de donner le « point de départ » de la génération « aléatoire » des blocks. L'utilisation du même *seed* pour générer les blocs générera la même série de blocs mais utiliser deux *seeds* différents générera deux séries de blocs différentes.

L'évaluation finale des scores des participants se feront avec l'un des *seeds* suivants : seed1, seed2, seed3, seed4, seed5, seed6, seed7 et seed8.

L'injection de ce *seed* se fait dans le constructeur de l'objet « TetrisWrapper » que vous utilisez pour démarrer le jeu.

```
/// <summary>
/// Create a new Tetris wrapper
/// </summary>
/// <param name="_teamId">A unique identifier</param>
/// <param name="_nbBlocks">[Optional] Maximum number of blocks per game</param>
/// <param name="_seed">[Optional] Random seed so the blocks are always the same</param>
public TetrisWrapper( String _teamId, Int32 _nbBlocks = 0, String _seed = null, GameSpeed
_gameSpeed = GameSpeed.Normal )
```

Aussi, vous avez la possibilité de changer le nombre de blocs par partie en changeant le paramètre « _nbBlocks » du constructeur. **Lors de l'évaluation finale, ce nombre sera toujours le même, 500**, mais rien ne vous empêche de la modifier pour vous aider dans le développement de votre solution.

Enfin, **l'évaluation finale, se fera à vitesse GameSpeed.Fast**

Score du jeu

Score attribué

Le score du jeu est calculé en fonction du nombre de lignes pleines que vous réussissez à faire disparaître. Aussi, **compléter 2 lignes d'un coup donne plus de points que compléter 2 lignes à des moments différents. 3 d'un coup c'est encore mieux. 4 c'est encore mieux.**

Les points affichés lors de la partie sont uniquement les points obtenus en complètant des lignes. Lorsque le jeu se termine, **le score est mis à jour en ajoutant 1 points par cases remplies par des blocs dans la grille**. Donc si deux équipes réussisent à complèter le même nombre de lignes avant d'être « Game Over », l'équipe ayant la grille avec le plus de cases remplies aura le meilleur score final.

Affichage du score

Pour faire afficher votre score sur le *dashboard* lors de vos essais, vous devez modifier la variable « TeamId » dans le fichier *Program.cs* de la solution Visual Studio avec le guid fourni dans le fichier TeamId.txt. Vous êtes obligés de le faire. Ça ajoute du piquant à la compétition de voir le score des autres équipes.

```
44      #region Constants
45
46      private const String TeamId = "Replace this value with your team id";
47
48      #endregion
```

Remise de la solution finale

À la fin du défi chaque équipe doit remettre aux juges sa clé USB qui contient sa solution Visual Studio et tous les fichiers nécessaires pour l'exécuter.

Évaluation

Le pointage affiché à l'écran géant n'est qu'informatif. **Seul le score de l'exécution finale faite par les juges sera considéré pour le calcul du pointage final.**

Le pointage final prend en compte le score obtenu avec votre solution ainsi que l'évaluation des juges.

La grille d'évaluation est notée sur 100 :

Le score final de la solution logicielle compte pour 50% de la note finale.

Le participant avec le plus haut score se verra attribuer la note maximale de 50 points pour le critère *Score final du jeu*. Les autres participants se verront attribuer une note proportionnelle à leur score en comparaison avec le meilleur score de jeu final (voir Annexe 1 pour exemple).

Critères	Note
Score final du jeu	50
Qualité du code - Propreté du code - Complexité cyclomatique (complexité faible = mieux) - Robustesse / Fragilité	25
Explication de la solution - Justification de la solution retenue (Pourquoi l'avez-vous choisie?) - Présentation de la solution (Comment l'avez-vous implémentée?)	20
Respect des règles	5
Total :	100

Pour remplir leur grille d'évaluation, les juges prendront des notes durant tout le défi. Ils passeront plus formellement voir chaque équipe à un moment donné durant les deux dernières heures du défi. Ils ne passeront qu'une seule fois pour l'évaluation formelle.

Autres règles

- Vous disposez de 4h45 pour réaliser le défi.
- Vous pouvez utiliser internet et son contenu.
- À l'exception des organisateurs de l'évènement, vous ne pouvez pas faire appel à de l'aide de tierces personnes. Vous ne pouvez pas non plus poser de question sur internet pour vous aider à résoudre le défi.
- Vous pouvez utiliser des logiciels et du code source de tierce partie tant que vous avez légalement le droit de l'utiliser et que cela n'enfreint aucune autre règle de ce document.

Vous devez participer de bonne foi. Il est notamment interdit de :

- Copier sur d'autres participants.
- Se faire aider par des personnes extérieures. Seuls les deux participants de l'équipe peuvent contribuer à l'élaboration de la solution.
- Désassembler ou faire de la « Réflexion » sur le code fourni. Il est cependant accepté de regarder le code interne que vous présente Visual Studio durant le débogage.
- Empêcher ou tenté d'empêcher l'affichage de son score au tableau des points.
- Écouter (« sniffer ») sur le réseau.
- Tenter de bloquer / saboter la soumission des solutions des autres équipes.
- Usurper l'identité d'autres équipes.
- Tenter de hacker les autres participants ou le réseau de l'entreprise.
- Faire quoi que ce soit d'illégal.

Tout manquement aux règles peut entraîner votre disqualification.

Annexes

Annexe 1 : Exemple de calcul de la traduction du score final du jeu en pointage d'évaluation

Participant A a un score de 100 000 points.

Participant B a un score de 75 000 points.

Participant C a un score de 60 000 points

Participant A se voit attribué la meilleur note possible pour la portion Score de jeu final, soit 50.

Participant B se voit attribué la note de ((75 000 / 100 000) * 50) = 37.5 pour la portion *Score de jeu final*

Participant C se voit attribué la note de ((60 000 / 100 000) * 50) = 30 pour la portion *Score de jeu final*

Annexe 2 : Formes des blocs et leur équivalent dans la solution Visual Studio

```
public enum TetrominoShape
  {
    Unknown = 0,
    /// Default configuration :
    ///I  _     <BR/>
    ///  | |   <BR/>
    ///  | |   <BR/>
    ///  | |   <BR/>
    ///  |_|   <BR/>
    I,

    ///J  _        <BR/>
    ///  | |___   <BR/>
    ///  |_____|  <BR/>
    J,

    /// L _____   <BR/>
    ///  |  ___|  <BR/>
    ///  |_|      <BR/>
    L,

    /// O ___    <BR/>
    ///  |   |   <BR/>
    ///  |___|   <BR/>
    O,

    /// S  _____    <BR/>
    ///  __|  __|   <BR/>
    /// |____|      <BR/>
    ///
    S,

    /// T __          <BR/>
    ///  |  |__       <BR/>
    ///  |   __|      <BR/>
    ///  |__|         <BR/>
    T,

    ///     Z ____          <BR/>
    ///       |__  |__       <BR/>
    ///          |____|      <BR/>
    Z,
  }
```

Annexe 3 : Structure de l'objet IState utilisé dans la solution Visual Studio

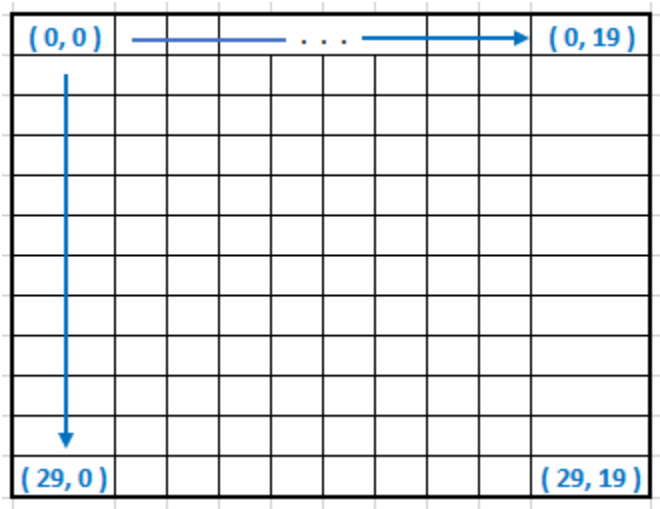
```
/// <summary>
/// An immutable representation of the game state at the begining of a tick. <BR/>
/// This version is NOT updated as the algorithm inputs commands. A new instance is generated each
/// tick. </summary>
public interface IState
{

  /// <summary>
  /// Corresponds to the currently playable piece. This piece is not present in the grid returned
  /// from the GetGrid() method.</summary>
  IPiece CurrentPiece { get; }

  /// <summary>
  /// Gets an enumerable of Boolean representing the row at index _index. Like the grid, the position
  /// containing a block will have a value of true and the position with no block will have a value
  /// of false.</summary>
  /// <param name="_index">The index of the row to get. The index ranges from 0 to 29</param>
  /// <returns></returns>
  IEnumerable<Boolean> GetRow( Int32 _index );

  /// <summary>
  /// Gets an enumerable of Boolean representing the column at index _index. Like the grid, the
  /// position containing a block will have a value of true and the position with no block will have
  /// a value of false.</summary>
  /// <param name="_index">The index of the column to get. The index ranges from 0 to 19</param>
  /// <returns></returns>
  IEnumerable<Boolean> GetColumn( Int32 _index );

  /// <summary>
  /// Returns whether the position ( x, y ) contains a block ( true ), or is free ( false )
  /// </summary>
  /// <param name="_x">The column index; ranges from 0 to 19</param>
  /// <param name="_y">The row index; ranges from 0 to 29</param>
  /// <returns>
  /// true if the position contains a block
  /// false of the position is free
  /// </returns>
  Boolean GetBlock( Int32 _x, Int32 _y );
```

```
  /// <summary>
  /// A reference to a grid showing whether each position is occupied by a piece or not.
  /// The current piece appears in it.   <BR/>
  /// Positions are accessed in X,Y order. So that the bottom right corner is GameGrid[19, 29]
  ///
  /// Grid coordinates :          <BR/>
  /// 0,0 -----------------> 0,19 <BR/>
  ///   |                         <BR/>
  ///   |                         <BR/>
  ///   |                         <BR/>
  ///   V                         <BR/>
  /// 29,0                        <BR/>
  /// </summary>
  Boolean[,] GetGrid();

  /// <summary>
  /// Gets the width of the grid. Spoiler alert, it's 19
  /// </summary>
  Int32 GridWidth { get; }

  /// <summary>
  /// Gets the width of the grid. Spoiler alert, it's 29
  /// </summary>
  Int32 GridHeight { get; }
}
```

Annexe 4 : Structure de l'objet IPiece utilisé dans la solution Visual Studio

```
  public interface IPiece: IEnumerable<Point>
  {
    /// <summary>
    /// The shape of the block. The possible shapes are I, J, L, O, S, T and Z
    /// </summary>
    TetrominoShape Shape { get; }

    /// <summary>
    /// Unique identifier
    /// </summary>
    Int32 Id { get; }

    /// <summary>
    /// [0-3] current rotation of the piece
    /// </summary>
    Int32 RotationIndex { get; }

    /// <summary>
    /// Positions of the blocks for the piece in the grid
    /// </summary>
    IEnumerable<Point> Blocks { get; }

    /// <summary>
    /// Simulate a rotation without actually moving the piece; used to know where the
current piece blocks would end if a rotation was made
    /// </summary>
    /// <param name="_rotationTimes">The number of rotation to simulate ( 1 rotation
turns the piece 90 degrees )</param>
    /// <returns>The simulated position of the rotated blocks</returns>
    IEnumerable<Point> SimulateRotation( Int32 _rotationTimes );
  }
```

Annexe 5 : Structure de la grille de jeu



Annexe 6 : Horaire de la journée

9:00 Arrivée des participants

9:30 Visite du bâtiment

9:45 Présentation du défi

10:15 Début de la programmation

4h45 de programmation. On mange en programmant.

15:00 Fin de la programmation

On ouvre pepsi/café + Ping pong/Hockey/Baby foot

Les juges délibèrent.

15:30 Cérémonie des prix

16:00 Départ