

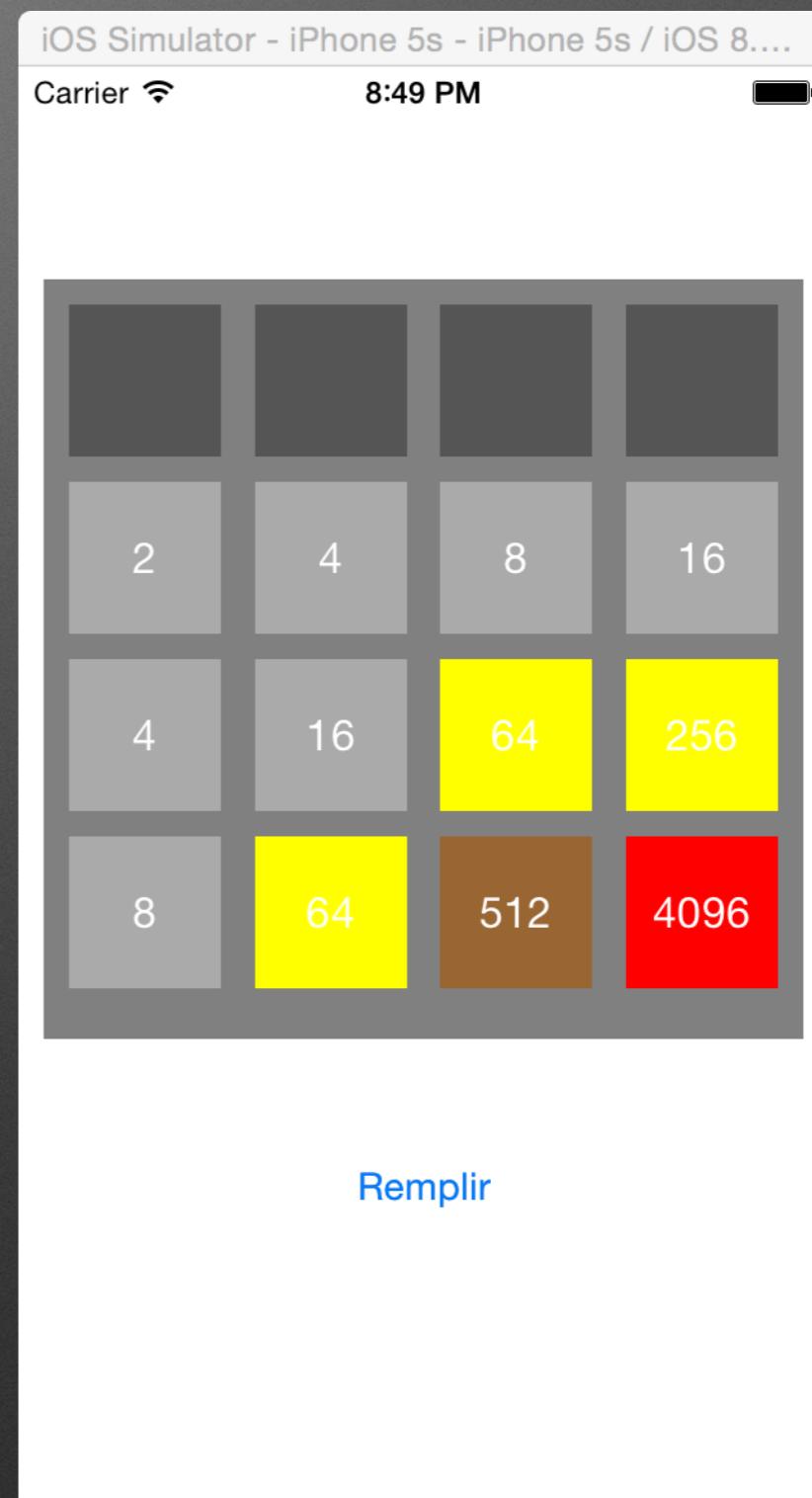
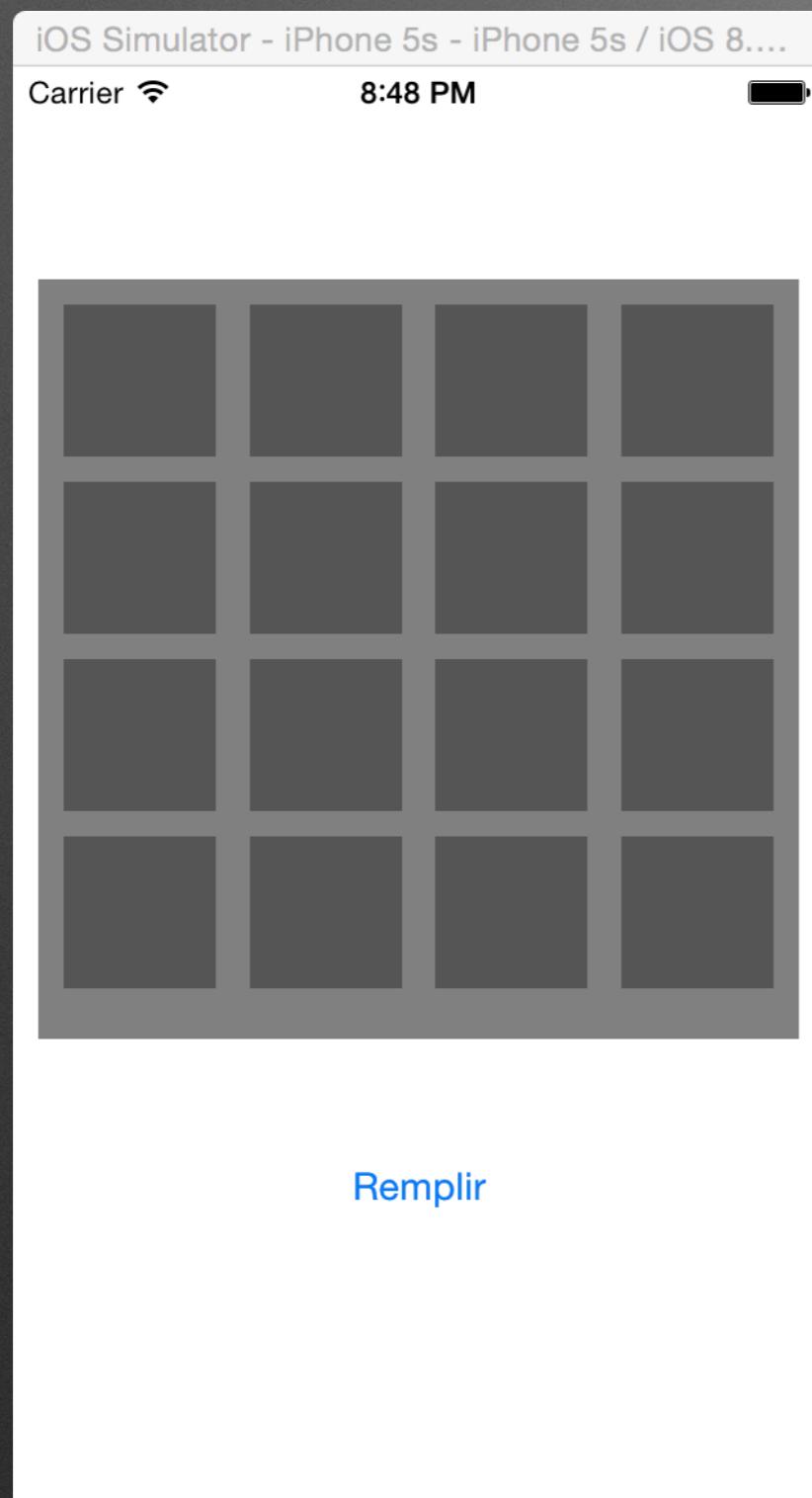


# Apprendre par la pratique la programmation mobile

Quelques aides pour le projet 2048

Anthony Fleury ([anthony.fleury@imt-lille-douai.fr](mailto:anthony.fleury@imt-lille-douai.fr))  
IMT Lille Douai - Année 2020 - 2021

# Dans ce pas à pas, vous allez apprendre à écrire l'interface pour le jeu de base...



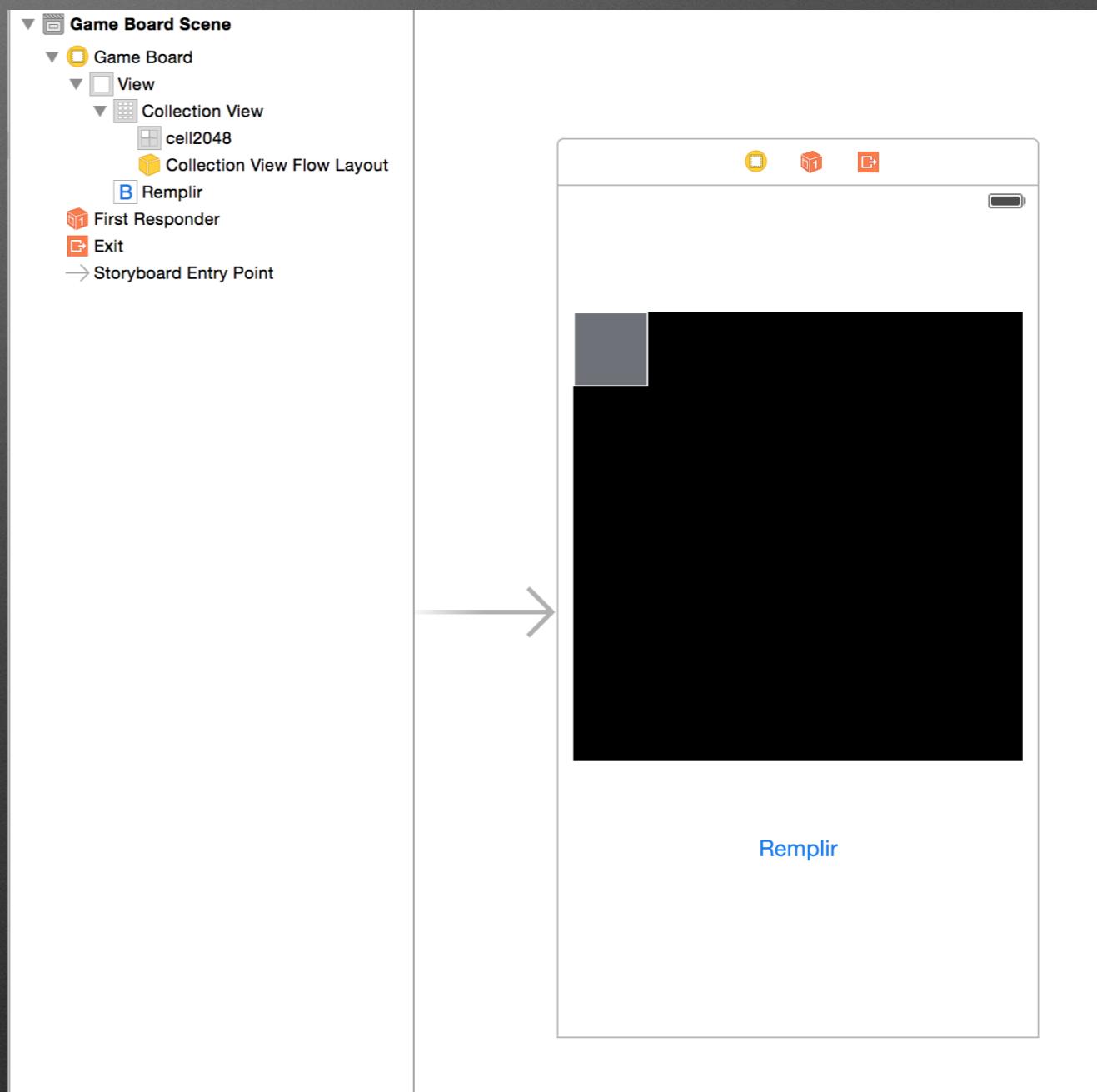
**Création de votre plateau de jeu et de la classe Jeu !**

# Commencer un projet

- Vous commencerez par créer un nouveau projet, en Swift.
- Dans ce projet vous allez avoir une classe gérant votre vue automatiquement créée et associée à votre vue dans storyboard.
- Nous allons jouer au départ sur cette vue et sur cette classe.
- Fixez-vous sur un modèle d'iPhone pour les tests et le développement.

# Vue et classe Jeu

- Dans votre vue, vous allez ajouter un contrôle de type « Collection View » qui va servir à gérer le jeu. Pour le début et pour tester ceci vous ajouterez également un bouton que vous nommerez « Remplir ».



# La classe associée à votre vue...

- Vous allez modifier la classe associée à la vue.
- Sur la première ligne, après le nom de la classe et le UIViewController, vous allez ajouter :  
« , UICollectionViewDataSource, UICollectionViewDelegate, UICollectionViewDelegateFlowLayout »
- Cela explique que vous allez suivre les protocoles (un ensemble de méthodes) de Collection View dans votre classe pour pouvoir construire cette vue collection. Votre classe va donc répondre à des messages de la ou des Collections View qu'elle possède.
- Ajoutez ensuite une variable de type UICollectionView! sous forme d'un IBOulet que vous allez associer à votre contrôle dans Storyboard.
- Ajoutez enfin deux variables de type entière pour le nombre de lignes et le nombre de colonnes. Donnez la valeur 4 comme valeur par défaut.

# Votre classe Cellule.

- Créez un nouveau fichier Swift qui va déclarer une classe pour gérer votre Cellule (l'un des éléments de votre vue).
- Ajoutez un import UIKit au début du fichier.
- La déclaration de votre classe ressemblera à ceci :  
`class GameCell: UICollectionViewCell {`
- Dans cette classe, vous allez ajouter :
  - une valeur entière qui sera la valeur de votre case (0 si aucune valeur). Sa valeur initiale sera 0.
  - une valeur booléenne, qui va indiquer si une cellule précise a déjà été impliquée dans un mouvement. Sa valeur initiale sera false.
  - un UILabel que vous n'associerez pas à storyboard mais gérerez vous même :

```
var texte: UILabel! = nil
```

# Ajout des constructeurs

- Ajoutez les deux constructeurs suivants à votre classe Cellule :

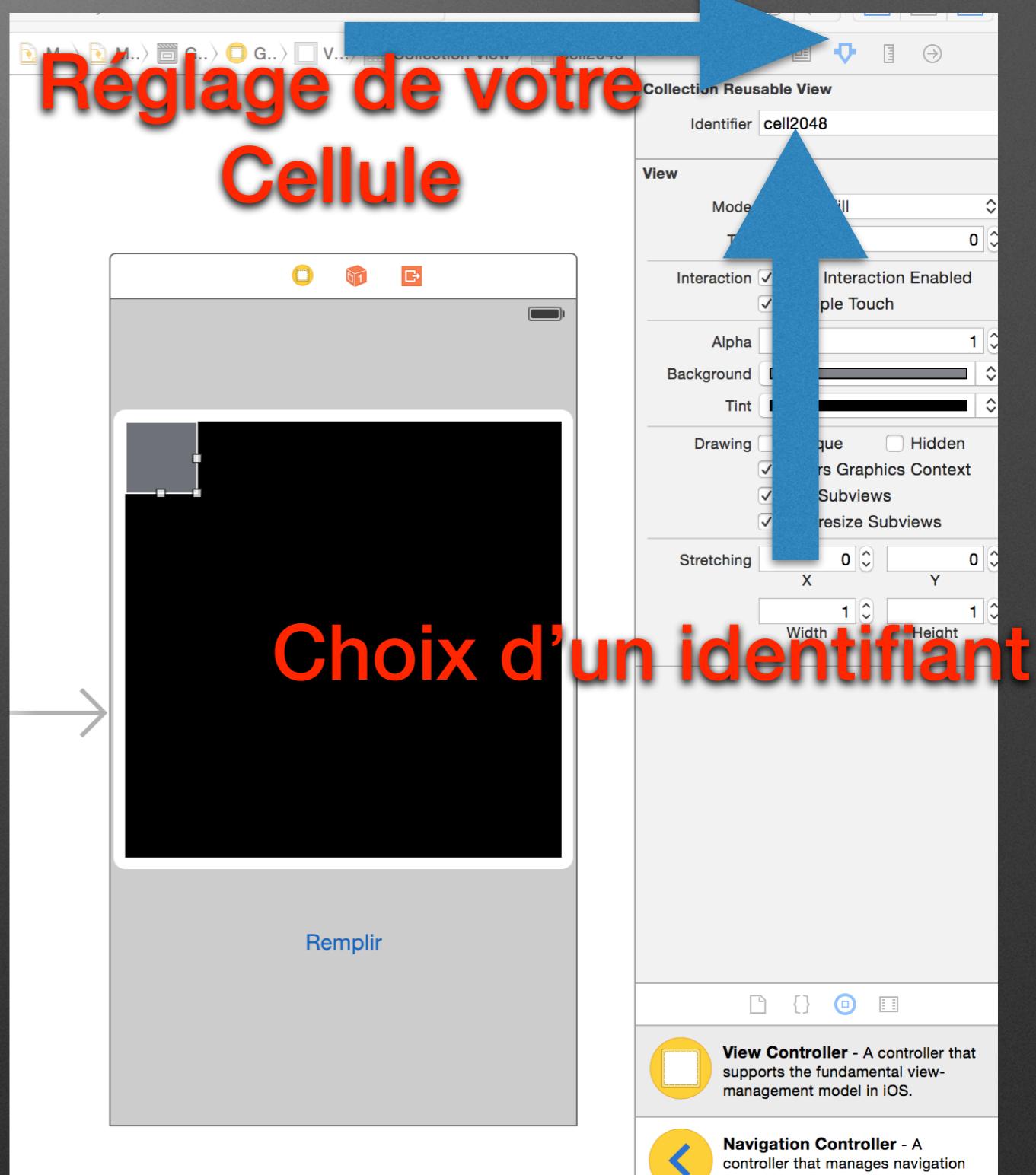
```
override init(frame: CGRect) {  
    super.init(frame: frame)  
}
```

```
required init(coder aDecoder: NSCoder) {  
    super.init(coder: aDecoder)  
}
```

- Ils sont obligatoires pour un bon fonctionnement de cette classe, même si ils ne font rien.

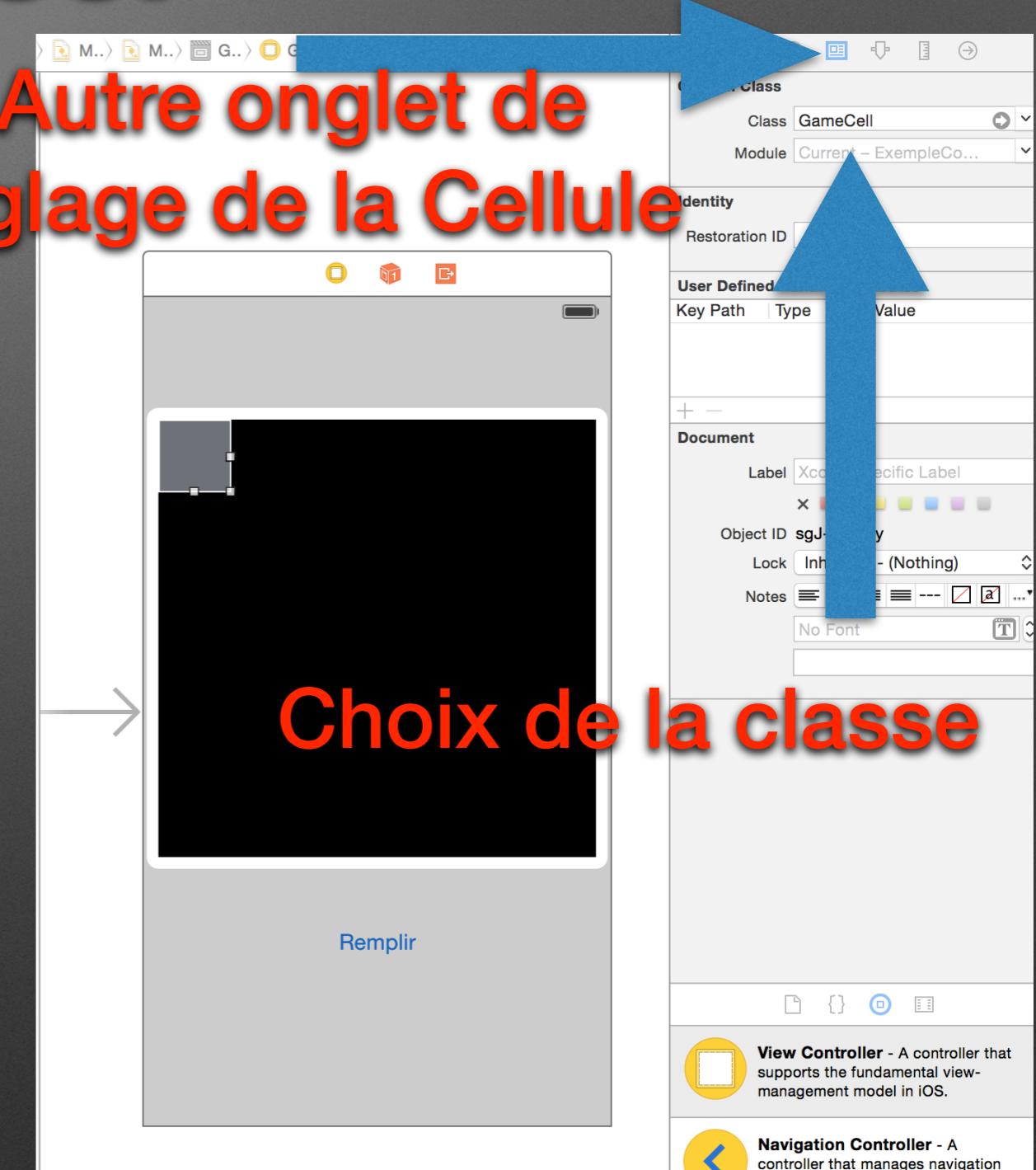
# Association de la cellule à la classe.

- Dans les réglages de votre cellule, vous allez donner un identifiant à votre cellule. Lorsque vous allez générer une nouvelle cellule, elle sera d'un des types donné et vous pourrez ainsi avoir différent paramétrages pour vos cellules.



# Association de la cellule à la classe.

- Associez ensuite votre cellule à la classe que vous avez créé précédemment.
- Lorsque vous créerez une nouvelle cellule de ce type, elle sera gérée par la classe créée.



# Retour sur votre classe Jeu.

- Revenons maintenant à la classe Jeu pour pouvoir construire l'affichage de nos cellules.
- Pour ceci, une méthode possible est par exemple de conserver un tableau à deux dimensions de cellules de jeu, optionnelles :
- `var cellules: [[GameCell?]]`
- Nous allons écrire l'initialiseur de notre classe et lui ajouter une initialisation de ces cellules avec un tableau de nil faisant la bonne taille :

# Retour sur votre classe Jeu.

- required convenience init?(coder aDecoder: NSCoder) {  
    self.init(coder: aDecoder, nombreLignes: 4, nombreColonnes: 4)  
}

```
init?(coder aDecoder: NSCoder, nombreLignes: Int, nombreColonnes:Int) {  
    self.nombreLignes = nombreLignes  
    self.nombreColonnes = nombreColonnes  
    cellules = []  
    cellules = ([[GameCell?]](repeating: [], count: nombreLignes-1))  
    for j in 1...nombreLignes {  
        let ligne = [GameCell?](repeating: nil, count: nombreColonnes)  
        cellules[j-1] = ligne  
    }  
    super.init(coder: aDecoder)  
}
```

- Ce constructeur va initialiser un tableau de cellules de type GameCell? de la bonne taille.

# Modification du viewDidLoad

- Dans votre viewDidLoad, vous pouvez ajouter du code permettant d'une part de générer les cellules, puis de régler quelques détails du collection view (la variable cells est votre UICollectionView) :

```
cells.delegate = self
cells.dataSource = self
let layout: UICollectionViewFlowLayout = UICollectionViewFlowLayout()

layout.sectionInset = UIEdgeInsetsMake(0, CGFloat(espaceCellules), 0,
                                      CGFloat(espaceCellules))
layout.minimumLineSpacing = CGFloat(espaceCellules)

cells.setCollectionViewLayout(layout, animated: false)

for i in 0...3 {
    for j in 0...3 {
        cellules[i][j] = cells.dequeueReusableCell(withIdentifier: "cell2048", for:
                                                    IndexPath(row: i, section: j) as IndexPath) as? GameCell
    }
}
cells.backgroundColor = UIColor.gray
```

# Faites que votre classe suive les protocoles...

- Vous devez maintenant écrire des méthodes permettant de répondre à certains messages concernant le Collection View...
- Une méthode renvoyant une cellule donnée et qui reçoit un indexPath (une position dans la collection sous forme de ligne/colonne) :

```
func collectionView(_ collectionView: UICollectionView,  
                  cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {  
    cellules[indexPath.section][indexPath.row]!.dessineCellule()  
    return cellules[indexPath.section][indexPath.row]!  
}
```

- Nous nous occuperons d'écrire la fonction dessineCellule() de notre cellule juste après... (celle appelée en première ligne de cette méthode)

# Faites que votre classe suive les protocoles...

- Une fonction renvoyant le nombre de sections (nombre de lignes) et le nombre d'item pour une section donnée (le nombre de colonne) :

```
func collectionView(_ collectionView: UICollectionView,  
                  numberOfItemsInSection section: Int) -> Int {  
    return nombreLignes  
}
```

```
func numberOfSectionsInCollectionView(collectionView:  
                                     UICollectionView) -> Int {  
    return nombreColonnes  
}
```

# Faites que votre classe suive les protocoles...

- Deux fonctions permettant de calculer la taille des cases et aussi de faire une présentation équilibrée (attention, pour ces deux fonctions, il faudra ajouter la déclaration de variable donnée juste avant) :

```
let espacementCellules = 10
func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout, referenceSizeForHeaderInSection section: Int) -> CGSize {
    return CGSize(width:0, height:CGFloat(espacementCellules))
}

func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath) -> CGSize {
    return CGSize(width: (cells.frame.width - CGFloat(espacementCellules * (nombreColonnes+2))) / CGFloat(nombreColonnes), height: (cells.frame.height - CGFloat(espacementCellules * (nombreLignes+2))) / CGFloat(nombreColonnes))
}
```

# Dessiner une cellule ?

- Nous allons ajouter, à la classe gérant les cellules, de quoi dessiner celle-ci :

```
func dessineCellule() {  
    if texte == nil {  
        texte = UILabel(frame: CGRect(x:0, y:0, width:self.bounds.size.width, height:self.bounds.size.height))  
        texte.numberOfLines = 1  
        texte.textAlignment = .center  
        texte.textColor = UIColor.white  
    }  
    switch valeur {  
        case let x where x >= 2 && x <= 16:  
            texte.text = "\((x))"  
            self.backgroundColor = UIColor.lightGray  
        case let x where x >= 32 && x <= 256:  
            texte.text = "\((x))"  
            self.backgroundColor = UIColor.yellow  
        case let x where x >= 512 && x <= 2048:  
            texte.text = "\((x))"  
            self.backgroundColor = UIColor.brown  
        case let x where x > 2048:  
            texte.text = "\((x))"  
            self.backgroundColor = UIColor.red  
        default:  
            texte.text = ""  
            self.backgroundColor = UIColor.darkGray  
    }  
    self.contentView.addSubview(texte)  
}
```

# Dessiner une cellule ?

- Vous noterez plusieurs choses dans cette méthode :
  - La première est qu'elle vous montre comment construire un contrôle à la main et le placer (le UILabel).
  - Ensuite, elle vous montre également comment régler des paramètres que vous pourriez régler dans l'interface graphique (la couleur du fond de la cellule etc.)
  - Enfin elle introduit une nouvelle notation, le switch case. Cette notation existe dans la plupart des langages. En swift elle a deux particularités : lorsque l'on rentre dans un cas on s'arrête ensuite. En java et dans tout autre langage, on continue jusqu'à la fin (fall through). Ensuite, contrairement aux autres langages aussi, la notation est plus puissante avec la clause where permettant de calculer des expressions.
  - Cette notation permet par exemple de réécrire le fizz buzz que vous avez vu en TP de Java de manière très élégante dans un seul switch case. Ceci vous est laissé en exercice !

# Association de votre bouton...

- Associez le bouton de votre interface graphique (permettant de remplir) à la fonction suivante que vous mettrez dans votre classe principale (celle gérant votre vue) :

```
@IBAction func rempli() {  
    for i in 0...3 {  
        for j in 0...3 {  
            cellules[i][j]!.valeur = Int(pow(2, Double(j+i*j)))  
        }  
    }  
}
```

- Ceci va remplir votre grille avec des valeurs en puissance de 2...
- Attention, pour que ceci fonctionne, il faut que vous ajoutiez un observer sur la valeur de votre cellule (avec didSet) pour appeler le dessin de votre cellule (mise à jour...).

# La gestion des événements tactiles...

- Pour gérer un évènement tactile, il suffit de s'enregistrer pour le suivre, en mettant un nom de méthode le gérant...
- Il y a plusieurs évènements que l'on peut reconnaître, et au final vous aurez toute liberté sur ceci. Mais pour commencer, nous pouvons gérer le « Swipe » (un déplacement) dans les 4 directions...
- Pour ceci, ajoutez le code suivant dans votre viewDidLoad :

```
let detectionMouvementR:UISwipeGestureRecognizer = UISwipeGestureRecognizer(target: self, action:  
#selector(self.mouvement))  
detectionMouvementR.direction = .right  
view.addGestureRecognizer(detectionMouvementR)  
let detectionMouvementL:UISwipeGestureRecognizer = UISwipeGestureRecognizer(target: self, action:  
#selector(self.mouvement))  
detectionMouvementL.direction = .left  
view.addGestureRecognizer(detectionMouvementL)  
let detectionMouvementH:UISwipeGestureRecognizer = UISwipeGestureRecognizer(target: self, action:  
#selector(self.mouvement))  
detectionMouvementH.direction = .up  
view.addGestureRecognizer(detectionMouvementH)  
let detectionMouvementB:UISwipeGestureRecognizer = UISwipeGestureRecognizer(target: self, action:  
#selector(self.mouvement))  
detectionMouvementB.direction = .down  
view.addGestureRecognizer(detectionMouvementB)
```

# La gestion des évènements tactiles...

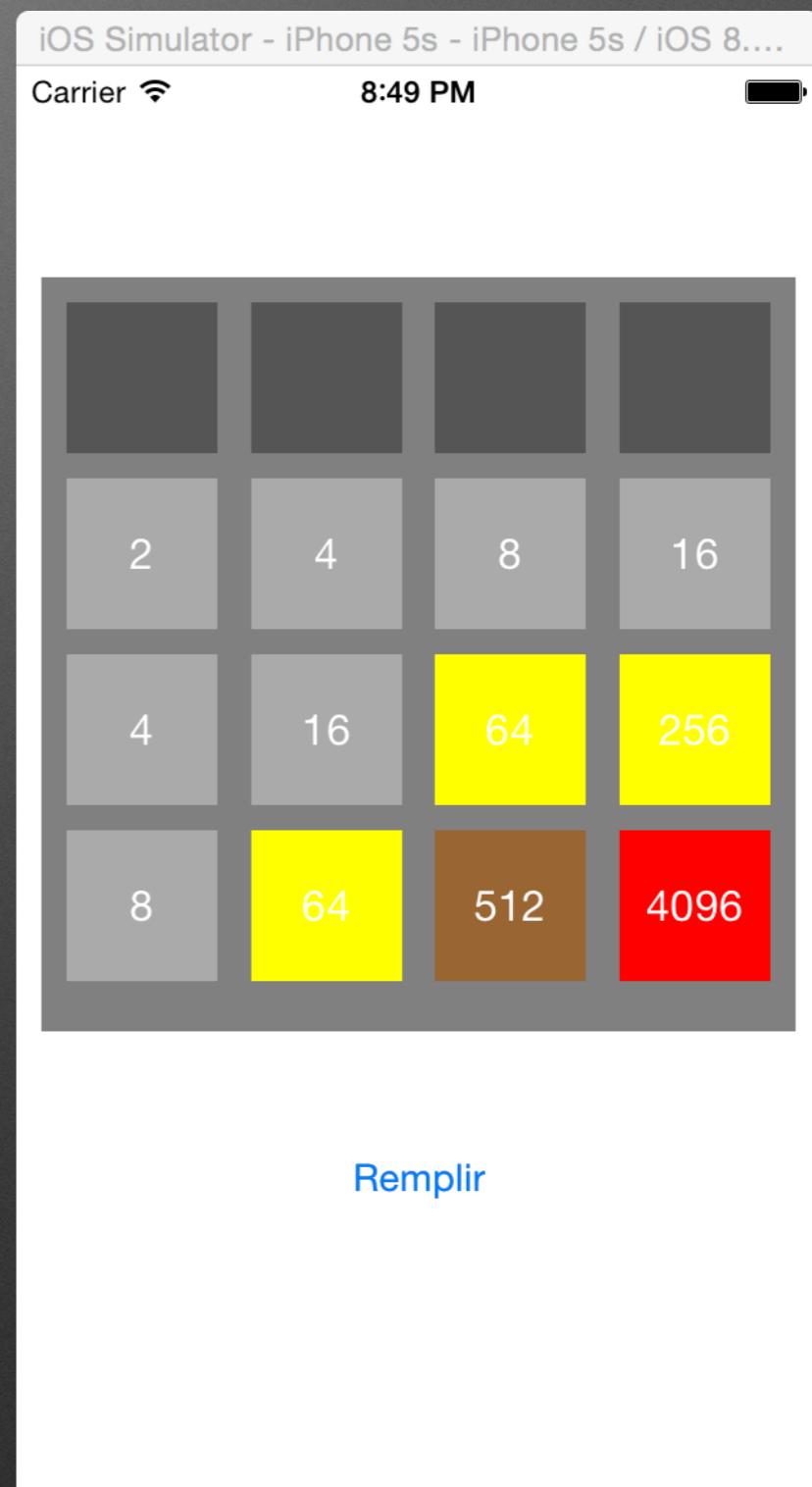
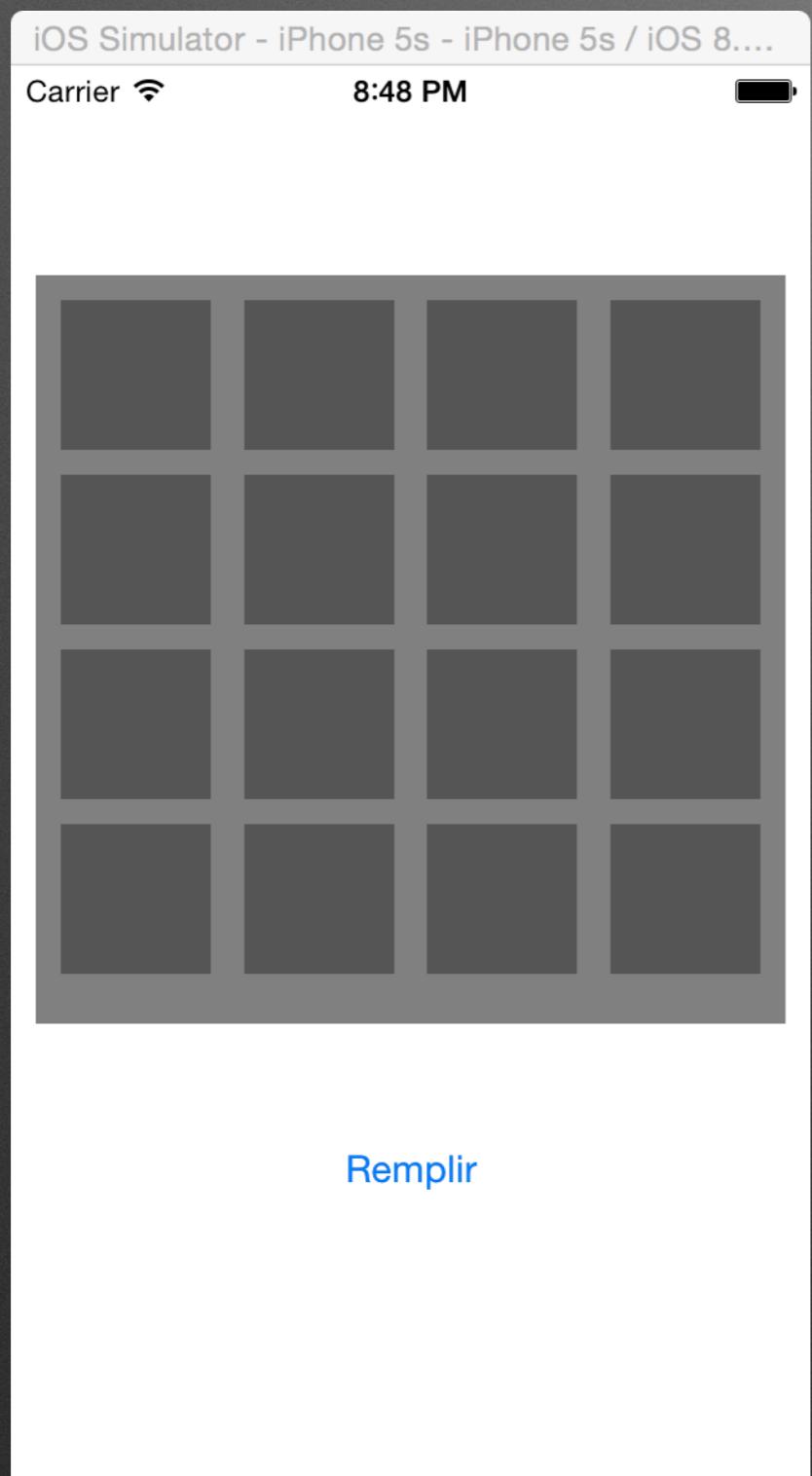
- Ce code (répétitif) permet d'enregistrer les 4 évènements et de les diriger vers une méthode de la classe en cours les gérant. Cette méthode est la suivante :

```
@objc func mouvement(sender:UISwipeGestureRecognizer){  
    switch sender.direction {  
        case UISwipeGestureRecognizerDirection.right:  
            print("Droite")  
        case UISwipeGestureRecognizerDirection.left:  
            print("Gauche")  
        case UISwipeGestureRecognizerDirection.up:  
            print("Haut")  
        case UISwipeGestureRecognizerDirection.down:  
            print(" Bas")  
        default:  
            break  
    }  
}
```

# La suite ?

- Il vous faut maintenant :
  - Ecrire une nouvelle classe (ou gérer cela dans une classe existante) pour gérer la règle du jeu. Cette classe sera appelée par la reconnaissance de geste avec les cellules en cours en paramètres pour gérer les gestes et le score.
  - Faire une plus jolie décoration pour votre tableau et mettre les couleurs et fonds de votre choix.
  - Gérer le fait de gagner une partie, savoir quand s'arrêter et gérer le remplissage de votre grille.
  - Gérer le fait de savoir si un mouvement est encore possible ou non (le joueur a-t-il perdu ?)

# Pour l'instant vous devez avoir :





# Apprendre par la pratique la programmation mobile

**Annexe : resize CollectionView**

Anthony Fleury ([anthony.fleury@mines-douai.fr](mailto:anthony.fleury@mines-douai.fr))  
IMT Lille Douai - Année 2016 - 2017

# Problème

- Logiquement les CollectionView se gèrent seules et font appel à des fonctions (nombre de sections, nombre de colonnes, item pour un IndexPath, etc.) pour se construire.
- Un appel à reloadData charge toutes les données.
- Or, dans les diapos précédentes, le choix fait pour gérer le plateau ne rend pas des plus simples une évolution. Ci-après un « correctif ».
- Ce correctif évite toute utilisation de InsertSections et InsertItemAtIndexPath source d'erreurs selon les ordres d'utilisation.

# Correction de vos « Items »

```
func collectionView(collectionView: UICollectionView,
    cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell
{
    if cellules[indexPath.section][indexPath.row] == nil {
        cellules[indexPath.section][indexPath.row] =
self.cells.dequeueReusableCellWithIdentifier("cell2048", forIndexPath:
NSIndexPath(forRow: indexPath.row, inSection: indexPath.section)) as? GameCell
    }
    cellules[indexPath.section][indexPath.row]!.dessineCellule()
    return cellules[indexPath.section][indexPath.row]!
}
```

- Cette correction propose de créer les cellules dans la méthode qui la renvoie si elle n'existe pas.

# Changer votre vue...

```
@IBAction func changer() {
    self.nombreColonnes = 6
    self.nombreLignes = 6

    self.cellules = [[]]

    self.cellules.appendContentsOf([[GameCell?]](count: self.nombreLignes-1,
repeatedValue: []))
    for j in 1...self.nombreLignes {
        let ligne = [GameCell?](count: self.nombreColonnes, repeatedValue: nil)
        self.cellules[j-1].appendContentsOf(ligne)
    }

    self.cells.reloadData()
}
```

- Ici on change le nombre de lignes et colonnes puis on recréé le tableau puis on rappelle reloadData. Chaque case du tableau sera de nouveau créée par la méthode changée précédemment (sinon les cases seraient nil).