

Support de cours

Cours:

## Introduction à la programmation orientée objet (en C++)

Vidéo:

### W11-05-morpions-CPP-pt1

Concepts (extraits des sous-titres générés automatiquement) :

**Interface d'utilisation d'une classe. Jeu du morpion. Utilisateur externe. Premier cas. Méthode d'initialisation de la grille. Second cas. Logique du jeu. Choix d'interfaces possibles. Programmeur de la classe jeumorpion. Tableau de taille fixe. Détails d'implémentation de la classe. Premier joueur. Type grille. Moyen d'un pointeur. Type de pièces.**



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en C++).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

# Encapsulation et abstraction : étude de cas

## (Partie 1)

Introduction à la programmation orientée objet (en C++)

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s





Bien penser l'interface d'utilisation d'une classe est une étape fondamentale

notes

---

---

---

---

---

---

---

---

---

---

résumé

0m 1s



---

---

---

---

---

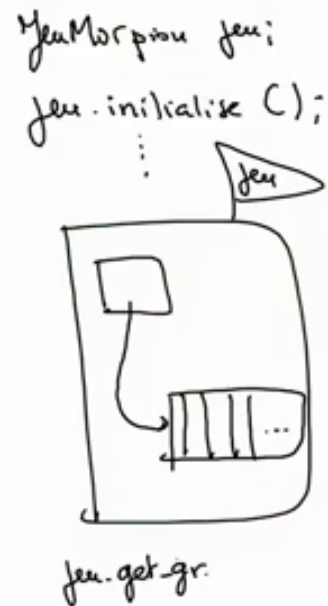
## Exemple : jeu de Morpion (1)

On veut coder une classe permettant de représenter le plateau 3x3 d'un jeu de Morpion (tic-tac-toe) :

```
typedef array<int, 9> Grille;

class JeuMorpion {
public:
    void initialise() { grille = new Grille; }
    Grille* get_grille() { return grille; }

private:
    Grille* grille;
};
```



dans toute bonne conception orientée objet. Une interface mal conçue qui offrirait trop de prise à l'utilisateur externe sur les détails d'implémentation, peut s'avérer tout aussi dangereuse que de laisser en public des attributs. Nous allons dans ce qui suit, comparer deux choix d'interfaces possibles pour une même classe. Dans un premier cas, l'interface offrira des fonctionnalités de trop bas niveau qui dévoilent à l'utilisateur externe les détails d'implémentation de la classe. Dans le second cas, l'interface n'offrira que ce qui est strictement nécessaire comme fonctionnalité. Et nous allons bien évidemment voir que la seconde option offre de nombreux avantages par rapport à la première. On s'intéresse ici à programmer une classe qui nous permette de jouer au jeu du Morpion, Tic Tac Toe. Donc pour rappel, dans le jeu du Morpion, deux joueurs s'affrontent sur une grille de 3 sur 3. 3 lignes, 3 colonnes. Le premier joueur peut poser un certain type de pièces, par exemple un rond. Le second joueur un autre type de pièces, par exemple une croix. Les deux joueurs jouent en alternance et le but pour gagner est d'essayer pour un des joueurs d'aligner 3 de ces pièces soit en ligne, soit en diagonale, soit en colonne. Dans la première version de la classe JeuMorpion que vous avez ici sous les yeux le programmeur a une vision assez minimaliste des services qu'il veut offrir au monde extérieur. Lorsqu'il a conçu sa classe, il s'est penché sur des problèmes de modélisation. Quelles sont les données qui caractérisent typiquement un jeu de morpion, et assez naturellement il s'est dit qu'il lui fallait une grille. Donc il a introduit un type Grille comme un tableau de taille fixe lui permettant de modéliser l'attribut grille de la classe JeuMorpion. En termes de fonctionnalités, accessibles depuis l'extérieur, il offre une méthode d'initialisation de la grille et il offre comme fonctionnalité principale, une méthode permettant à l'utilisateur externe d'accéder à

### notes

### résumé

0m 5s



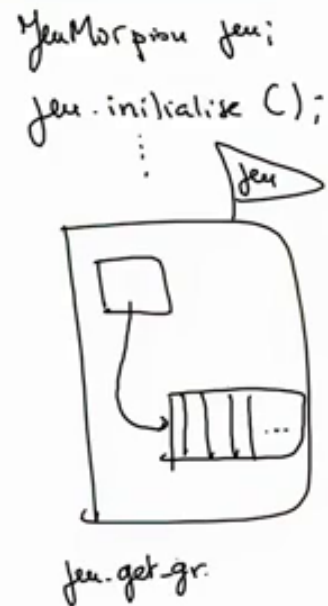
## Exemple : jeu de Morpion (1)

On veut coder une classe permettant de représenter le plateau 3x3 d'un jeu de Morpion (tic-tac-toe) :

```
typedef array<int, 9> Grille;

class JeuMorpion {
public:
    void initialise() { grille = new Grille; }
    Grille* get_grille() { return grille; }

private:
    Grille* grille;
};
```



la grille de jeu. Donc un utilisateur qui accéderait à cette grille de jeu depuis l'extérieur aurait toute l'attitude de pouvoir y placer à sa guise des jetons pour implémenter la logique du jeu à proprement parler. Examinons maintenant de plus près les détails d'implémentation de cette première version. Tout d'abord, qu'en est-il du type Grille qui permet de modéliser une grille de jeu ? Si l'on examine le type associé, on se rend compte ici que le programmeur a fait un choix particulier qui n'est peut-être pas très naturel mais qui est possible d'en faire un tableau à 1 dimension. Ce qui veut dire que probablement il voit les choses de cette façon-là. Donc la grille de jeu est modélisée par le biais d'un tableau de taille fixe à 1D où par exemple les 3 premières positions représentent la première ligne de la grille du jeu de morpion ; les 3 suivantes représentent la seconde ligne et les 3 dernières la dernière ligne. Donc c'est un choix possible même si à priori, il n'est peut-être pas très naturel. Le programmeur de la classe JeuMorpion souhaite offrir à l'utilisateur de cette classe, un accès à la grille afin que cet utilisateur puisse remplir par lui-même la grille. Il fait donc le choix de modéliser la grille au moyen d'un pointeur sur une grille. Et c'est ce pointeur qui est offert au travers de la fonctionnalité `get_grille`. Concrètement donc, notre attribut grille n'est autre qu'un pointeur vers un tableau de taille fixe à 9 cases. Comme fonctionnalité principale, nous avons vu que le programmeur de la classe JeuMorpion offrait la méthode `get_grille` laquelle permet à l'utilisateur externe d'accéder à la valeur de ce pointeur. Par le biais de ce pointeur il va être possible à celui qui utilise la classe, d'accéder à la grille et de la remplir pour implémenter la logique du jeu. Donc on voit que dans cette conception,

### notes

### résumé

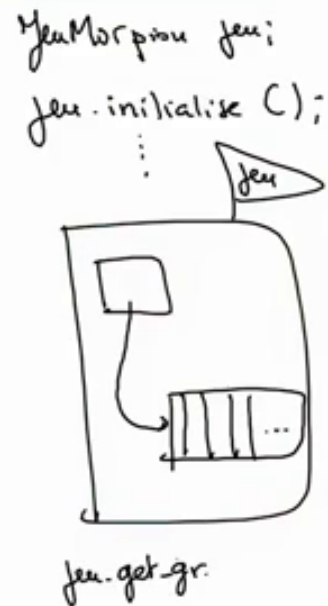
## Exemple : jeu de Morpion (1)

On veut coder une classe permettant de représenter le plateau 3x3 d'un jeu de Morpion (tic-tac-toe) :

```
typedef array<int, 9> Grille;

class JeuMorpion {
public:
    void initialise() { grille = new Grille; }
    Grille* get_grille() { return grille; }

private:
    Grille* grille;
};
```



le programmeur de la classe `JeuMorpion` délègue la programmation de la logique du jeu à celui qui utilise la classe plutôt que de la placer à l'intérieur de la classe `JeuMorpion`. Ce choix est évidemment très mauvais et nous aurons l'occasion d'y revenir. La seconde méthode offerte par le programmeur de la classe `JeuMorpion` est la méthode `initialise` qui se contente ici d'allouer un espace mémoire pour un tableau d'entiers à 9 cases. Donc quelqu'un qui souhaiterait utiliser la classe `JeuMorpion` pourrait le faire de la façon suivante ; Donc il commencerait par déclarer une variable `jeu` par exemple de type `Jeu Morpion`. Ensuite il pourrait appeler la méthode `initialise...` pour initialiser la grille. Au terme de cette initialisation nous aurions donc, dans la variable `jeu`, un objet dont l'un des attributs est un pointeur, une adresse vers un tableau de 9 cases. Pour programmer ensuite la logique du jeu, c'est par exemple cet utilisateur qui va programmer la logique qui consiste à faire jouer en alternance l'un ou l'autre des joueurs plaçant des ronds et des croix. Eh bien le programmeur utilisateur de la classe `JeuMorpion` va devoir accéder via ce pointeur à la grille et y placer des ronds et des croix selon ses besoins. Pour connaître la valeur de ce pointeur lui permettant justement d'accéder à la grille de jeu et la remplir, l'utilisateur de la classe `JeuMorpion` va utiliser la fonctionnalité offerte publiquement `get_grille` Donc ici, pour accéder à la valeur de ce pointeur, il suffit d'utiliser

### notes

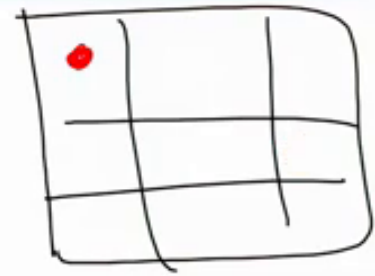
### résumé

## Exemple : jeu de Morpion (2)

```
typedef array<int, 9> Grille;

class JeuMorpion {
public:
    void initialise() { grille = new Grille; }
    Grille* get_grille() { return grille; }

private:
    Grille* grille;
};
```



Le joueur rond coche la case en haut à gauche :

```
JeuMorpion jeu;
jeu.initialise();
(*jeu.get_grille())[0] = 1;
```

Convention : 1 représente un rond, 2 une croix et 0 une case vide

la fonctionnalité `get_grille` comme ceci. Et nous verrons concrètement comment on va utiliser cette fonctionnalité justement pour remplir la grille en utilisant les fonctionnalités offertes par cette première version de la classe `JeuMorpion`. Voilà, plaçons nous maintenant dans la situation où un programmeur-concepteur a offert une classe `JeuMorpion` avec les fonctionnalités que nous venons de décrire et que maintenant un programmeur-utilisateur souhaite en faire usage pour programmer par exemple, la logique du jeu de morpion. Imaginons que ce programmeur-utilisateur s'intéresse à la question suivante : comment placer concrètement, sur la grille du jeu de morpion, par exemple un rond dans la première case de la grille ? La première chose qu'il va devoir faire probablement est d'adopter un certain nombre de conventions sur la représentation du contenu de ces différentes cases. Sachant que le tableau représentant la grille est un tableau d'entiers, il peut par exemple décider d'un certain nombre de conventions comme par exemple que le "1" représente un rond, que le "2" représente une croix et le "0" une case vide. Ceci pose au passage un certain nombre de questionnements sur l'initialisation de la grille. Si on examine le code, le concepteur de `Jeu Morpion` s'est contenté d'allouer l'espace mémoire pour le tableau associé à la grille, mais à nul moment il ne s'est préoccupé d'initialiser le contenu de chacune de ces cases, qui ici est indéterminé. Si par exemple l'utilisateur de `Jeu Morpion` décide comme il se doit de commencer avec une grille entièrement vide, c'est à lui de parcourir la grille et de la remplir avec des 0 pour signifier qu'il y a des cases vides partout pour commencer. Ici, le travail d'initialisation devrait clairement être pris en charge par celui qui offre la grille et non pas par celui qui l'utilise. et non pas par celui qui l'utilise.

### notes

### résumé

5m 13s

