Support de cours

Cours:

Initiation à la programmation (en Java)

Vidéo:

Init-JAVA-07-3-joue1-pt4

Concepts (extraits des sous-titres générés automatiquement) :

Variable ligne. Position vide. Dernière ligne du tableau. Condition vraie. Nouvelle fonction main. I égal. Colonne correspondante. Boucle while. Exemple des pions rouges. Variable booléenne. Première version de la méthode. Colonne. Partie else. Case vide. Valeur de la variable booléenne pleine.

vers la recherche de séquences vidéo
(dans Initiation à la programmation (en Java).)

vers la vidéo

EPFL

Puissance 4 : méthode `joue` 1ère version
(Partie 4)

Initiation à la programmation (Java)

Jamila Sam, Vincent Lepetit et Jean-Cédric Chappelier

...

notes

résumé

0m 0s

Méthode joue

```
static void joue(int[][] grille, int colonne, int couleur)
{
  // on parcourt la colonne en partant du bas jusqu'à trouver une case vide,
  // ou jusqu'en haut de la colonne si la colonne est pleine :
  int ligne = grille.length - 1;

  boolean pleine = false;
  while ((!pleine) && (grille[ligne][colonne] != VIDE)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jus
  if (!pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
```

Maintenant, que se passe-t-il si on continue à jouer comme ceci dans la colonne 3, et si on rajoute des pions, par exemple des pions rouges chaque fois dans la colonne 3, que va faire notre méthode joue dans ce cas-là ? Que se passe-t-il quand on joue un pion dans une colonne qui est déjà remplie ? Ce qui va se passer c'est que l'on initialise comme d'habitude notre variable ligne à la dernière ligne du tableau, puis tant que la ligne, à la colonne correspondante n'est pas vide, en l'occurrence, ici, jusqu'à ce que la ligne atteigne la position zéro, cette condition va être vraie, on va toujours faire "--ligne ", on va remonter comme ça jusqu'à la position zéro. Quand ligne vaut zéro, alors qu'est-ce qu'il se passe ? À ce moment là, on va avoir cette condition qui est toujours vraie, la position zéro n'est pas vide, et on recommence à faire un "--ligne ", ce qui fait qu'on va comme ça, s'enfuir au-delà de notre tableau, au-delà de notre jeu, en tout cas c'est une erreur de conception, puisqu'on a ici une boucle qui nous fait sortir du tableau. Donc, il va falloir corriger ceci, et arrêter cette boucle while, dès que l'on dépasse la fin du tableau, c'est-à-dire dès qu'on atteint l'indice zéro. Il ne reste plus qu'à l'écrire, pour cela on va déclarer une variable booléenne qui va tester si on a remplit complètement une colonne, donc on va l'appeler typiquement " pleine ". Au départ, à priori, la colonne n'est pas supposée être pleine, donc on va déclarer cette variable à false. Et puis, dans la boucle qui va remonter pour rechercher une position vide, on va changer la valeur de la variable booléenne pleine, et la faire passer à vraie, dès que l'on

notes

résumé

0m 1s

Méthode joue

```
static void joue(int[][] grille, int colonne, int couleur)
{
  // on parcourt la colonne en partant du bas jusqu'à trouver une case vide,
  // ou jusqu'en haut de la colonne si la colonne est pleine :
  int ligne = grille.length - 1;

  boolean pleine = false;
  while ((!pleine) && (grille[ligne][colonne] != VIDE)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jus
  if (!pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
```

atteint la position zéro, puisque si dans cette boucle, on arrive jusqu'à la position zéro, c'est que la position zéro n'était pas vide, donc la position zéro n'est pas vide, que si le reste de la colonne est plein. Donc, continuons à l'écrire, si la ligne atteint la position zéro, à ce moment-là, on dit que la colonne est pleine, donc " pleine = true ". Dans ce cas-là, en plus, il ne faut pas décrémenter, il ne faut pas soustraire un à la ligne, et donc on protège notre instruction "--ligne ", ici par le fait que la ligne n'est pas nulle, donc on met ici une partie else qui protège cette décrémentation, ce "--ligne " de la condition, du fait que la ligne soit nulle. Et on corrige maintenant notre boucle, en disant qu'on continue la boucle, comme tout à l'heure, c'est-à-dire à rechercher une position vide, mais tant que la colonne n'est pas pleine. Ce signe là, je vous rappelle que ça veut dire not, donc tant que la colonne n'est pas pleine, et qu'on a pas trouvé une case vide, on continue tant que les 2 conditions sont vraies. Si elle n'est pas pleine et si on a pas trouvé une position vide,on remonte. Si jamais l'une des deux conditions, soit qu'elle est pleine, soit qu'on a trouvé une case vide, était vraie, à ce moment là, la boucle s'arrêterait. On peut terminer en ajoutant ici une paire de parenthèses pour rendre notre code plus lisible. Enfin, puisqu'on ne l'a pas fait au début, ce qu'on aurait du faire, on commente ce que l'on vient de faire, L'idée c'est qu'on parcours la colonne en partant du bas jusqu'à trouver une case vide, ou jusqu'en haut. Enfin, il faut ne pas oublier de corriger la fin qui n'est plus valide, puisqu'on

notes

résumé

Méthode joue

```
static void joue(int[][] grille, int colonne, int couleur)
{
  // on parcourt la colonne en partant du bas jusqu'à trouver une case vide,
  // ou jusqu'en haut de la colonne si la colonne est pleine :
  int ligne = grille.length - 1;

  boolean pleine = false;
  while ((!pleine) && (grille[ligne][colonne] != VIDE)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jus
  if (!pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
```

ne peut plus à chaque fois garantir que l'on peut jouer le pion, puisque si la colonne est pleine, on ne joue pas le pion. Donc on protège le fait le jouer, par la condition : si la colonne n'est pas pleine, donc " if (!pleine) ", à ce moment là, on peut jouer, et on peut en même temps donner une indication : notre fonction joue, maintenant, on va pouvoir lui donner une indication à l'extérieur, et par exemple, renvoyer true, pour dire : " oui j'ai bien pu jouer, la fonction joue a pu faire ce que l'on veut ". Et puis, sinon, par exemple, retourner false, pour dire : " non, la colonne était pleine, le coup n'était pas valide " " et je n'ai pas pu placer mon pion à la bonne position ". Enfin, comme d'habitude, on commente notre code pour expliquer, soit à nous plus tard, qui allons le reprendre,

notes

résumé

soit à d'autres personnes qui le liraient, ce que fait notre code. Ici si on n'est pas arrivés jusqu'en haut, si " if (!pleine)", si on est pas arrivés jusqu'en haut de la colonne, à ce moment là, on peut jouer. N'oublions pas, enfin, de corriger notre type de retour, puisque nous avons maintenant décidé de renvoyer un booléen qui nous indique si le coup était valide ou non, donc on change, ici le type de retour de void à un booléen.

notes

résumé

4m 37s

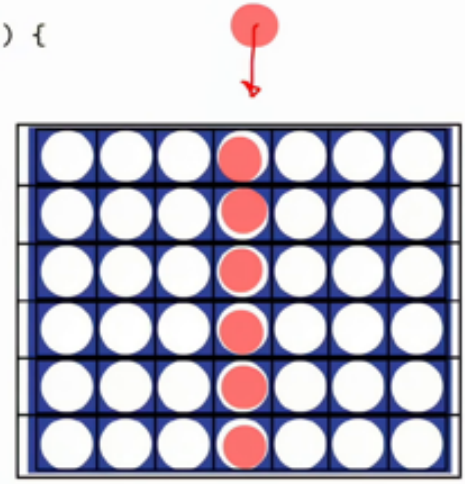
Tester la nouvelle méthode joue

```
static boolean joue(int[][] grille, int colonne, int couleur)
{
  int ligne = grille.length - 1;
  boolean pleine = false;
  while ((!pleine) && (grille[ligne][colonne] != VIDE)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  if (!pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
...
  for(int i = 0; i < 10; ++i) {
    boolean valide = joue(grille, 3, ROUGE);
    if (!valide) {
      System.out.println("impossible d'ajouter un pion sur cette colonne");
    }
    affiche(grille);
  }
```

3 Rouge

0 →



Voilà, nous pensons donc maintenant avoir corrigé notre première version de la méthode joue en une version qui est robuste au fait que la colonne est pleine, et donc il faut maintenant tester cette méthode, et pour ça, on va réécrire une nouvelle fonction main, où on va tester spécifiquement le fait de jouer tout le temps dans la même colonne, comme d'habitude, on déclare la grille, on l'initialise, on l'affiche, et puis ici, on va faire une boucle qui va jouer 10 fois dans la même colonne, ici toujours dans la quatrième colonne, la colonne numéro 3, on fait une boucle qui fait 10 itérations, qui joue dans la colonne 3. Remarquez que maintenant, la fonction joue retournant un booléen qui indique si le coup était valide, on appelle maintenant joue en partie droite d'une expression, on récupère donc la valeur, on écrit " valide = joue(...)", et puis pour voir ce qui se passe et pour tester notre programme, si le coup n'était pas valide, alors on va afficher un message qui nous dit qu'il n'était pas possible de jouer. Dans tous les cas, on affiche la grille, pour voir ce qui se passe. Qu'est-ce que ça va nous donner ? J'ai recopié ici le main en bas, et toute notre méthode joue. Exécutons le main, on part à partir de " i = 0 ", on n'utilise pas le i, c'est juste pour dire qu'on va faire 10 fois la boucle. Donc on part ici de " i = 0 ", on joue toujours dans la colonne 3 et en rouge, Ça va commencer à jouer ici un pion, puis ça va jouer un autre pion, etc. Tous ces cas là ne sont pas spécialement intéressants, mais que va-t-il se passer quand on va jouer le septième coup ? Une fois

notes

résumé

5m 5s

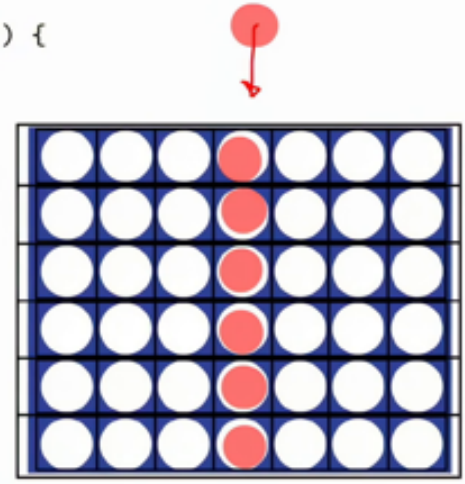
Tester la nouvelle méthode joue

```
static boolean joue(int[][] grille, int colonne, int couleur)
{
  int ligne = grille.length - 1;
  boolean pleine = false;
  while ((!pleine) && (grille[ligne][colonne] != VIDE)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  if (!pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
...
  for(int i = 0; i < 10; ++i) {
    boolean valide = joue(grille, 3, ROUGE);
    if (!valide) {
      System.out.println("impossible d'ajouter un pion sur cette colonne");
    }
    affiche(grille);
  }
```

3 Rouge

0 →



que i vaut 6, ici, on a envie de jouer ici un septième pion qu'on essaie de faire venir dans la colonne qui est pleine, et donc quand i vaut 6, on va rentrer à nouveau ici, avec la valeur 3 pour la colonne et puis la couleur rouge, et on va, ici, initialiser la ligne à la taille de la grille moins un. On va initialiser booléen à faux, donc évidemment, notre pleine est vrai, la grille, à la position [ligne][colonne], n'est pas vide, la ligne n'est pas égale à zéro, donc on saute ici dans le else, et on décrémente ligne, qui va donc passer ici (2ème ligne). Ici, on a toujours les 2 conditions qui sont vérifiées, donc on a toujours notre pleine qui est vraie, et on a toujours une grille, ici, qui n'est pas vide, et donc et cetera, on va monter jusqu'à arriver à la position ligne==0. Donc quand ligne==0, que va-t-il se passer ? On arrive toujours ici avec une condition vraie, et la grille de [ligne], de zéro, de [colonne], n'est pas vide, on rentre, mais cette fois-ci, le if est vrai, la ligne vaut zéro, à ce moment là on change et on met la valeur de pleine à vraie.

notes

résumé

On est ici, on a donc terminé le bloc du if, et on revient une nouvelle fois dans notre boucle, mais cette fois-ci, la condition, ici "!pleine " est fausse, et donc, le while s'arrête, et on termine, on sort de la boucle. Ensuite on continue, ici on arrive sur ce if, de nouveau, notre pleine, ici, n'est pas vrai, donc cette condition, ici, est fausse, on saute dans le else et on retourne false. Ce qui fait que, arrivé, ici, quand if vaut 6, valide va valoir faux, et donc, on va afficher le message d'erreur, et donc ce message d'erreur s'affichera pour i égal 6, 7, 8 et 9, on aura donc 4 fois un affichage du message de l'impossibilité de jouer. Voilà donc le test qui nous confirme qu'on ne peut pas remplir une colonne qui était déjà pleine, c'est exactement ce que nous voulions. Voilà, je propose de terminer ici cette séquence vidéo, et d'aborder dans une séquence suivante des versions alternatives qu'on aurait pu choisir pour implémenter la méthode joue. pour implémenter la méthode joue.

notes

résumé

8m 13s