PROGRAMMATION ORIENTÉE OBJET

Informatique de gestion - bloc 2

Haute-École de Namur-Liège-Luxembourg

Laboratoire 2 : propriétés et paramètres

Objectifs

- Découvrir, comprendre et savoir manipuler la notion de propriétés
- Se familiariser avec diverses options pour la gestion des paramètres des méthodes
- Appliquer toutes ces connaissances dans le cadre d'exercices pratiques

Exercice 1 : Membres d'un forum

L'objectif de cet exercice est de construire une classe `User` permettant de représenter les membres inscrits à un forum.

Étape 1 : les fondations de la classe User

Dans un nouveau projet, créez une classe `User`. Les attributs à encoder sont les suivants :

- `login` : nom d'inscription du membre (chaîne de caractères)
- `password` : mot de passe du membre (chaîne de caractères, du moins dans un premier temps)
- `joinDate` : date d'inscription (un objet de type `DateTime`)
- `postCount` : nombre de messages écrits (un entier, initialement à 0 lors de la création de l'objet – notez que cela correspond à l'effet de l'initialisation automatique)

En plus de la déclaration des attributs (tous privés), ajoutez les éléments suivants.

- Constructeur recevant le login, le mot de passe et la date d'inscription (le `postCount` sera initialisé à 0)
- Constructeur recevant le login et le mot de passe, faisant appel au premier constructeur et utilisant la date d'aujourd'hui (qu'on peut obtenir via l'écriture `DateTime.Today`)
- Méthode `AddPost()` permettant d'incrémenter le nombre de messages d'un membre
- Méthode `ToString()` renvoyant une chaîne selon le format suivant (voir note) :

```
Herbert (password: 12345), 23-Oct-17 00:00:00 - 0 post
Herbert (password: 12345), 23-Oct-17 00:00:00 - 1 post
Herbert (password: 12345), 23-Oct-17 00:00:00 - 2 posts
```

Dans la méthode `Main` de la classe `Program`, rédigez quelques lignes de tests en créant et en affichant deux ou trois membres. Pour rappel, comme en Java, l'appel à la méthode `ToString()` peut souvent être sous-entendu.

Note. Faites une recherche sur Google selon les mots-clefs « C# DateTime » pour trouver de la documentation au sujet de ce type vous permettant de trouver une méthode qui donne directement la date au format demandé.

Étape 2 : Getter, setter et classe utilitaire

Ajoutez à la classe un getter `GetLogin()` et un setter `SetLogin(string newLogin)` permettant d'obtenir et de modifier le nom de login.

Dans le cas du setter, on devrait vérifier que le nouveau login n'est pas déjà utilisé et qu'il correspond aux règles en vigueur sur le forum. Pour éviter de polluer la classe `User`, créez une nouvelle classe appelée `ForumUtils` et déclarez-la avec le modificateur `static`, qui indique que cette classe ne possédera aucune instance mais uniquement des méthodes de classe (= méthodes utilitaires).

Dans la classe `ForumUtils`, déclarez une constante reprenant les logins interdits (vous pouvez éventuellement ajouter des noms à la liste).

```
public static string[] FORBIDDEN_LOGINS = new [] { "", "fart", … };
```

Ajoutez à `ForumUtils` une méthode de classe `ValidLogin (string login)` indiquant si le login proposé est un login valide.

Pour tester votre implémentation, utilisez les deux méthodes suivantes (que vous pourrez ajouter à la classe `Program`) en appelant `TestValidLogin()` dans la méthode `main`.

```
static void AssertBool (string test, bool expected, bool observed)
{
  Console.WriteLine("Test: " + test);
  Console.WriteLine("Expected: " + expected + ", observed: " + observed);
  Console.WriteLine(expected == observed ? "Ok!" : "KO !!!");
  Console.WriteLine();
}

static void TestValidLogin()
{
  AssertBool("Herbert", true, ForumUtils.ValidLogin("Herbert"));
  AssertBool("empty string", false, ForumUtils.ValidLogin(""));
  AssertBool("fart", false, ForumUtils.ValidLogin("fart"));
  AssertBool("FART", false, ForumUtils.ValidLogin("FART"));
  AssertBool("FaRt", false, ForumUtils.ValidLogin("FaRt"));
}
```

Notez que, pour obtenir une copie en minuscules d'une chaîne de caractères donnée, vous pouvez utiliser la méthode `.ToLower()`.

Note - le format des méthodes de tests

Pour réaliser un test, vous vous contentez sans doute généralement d'écrire un appel à la méthode à tester et d'afficher le résultat, puis de lancer le programme et de vérifier le résultat affiché. Cela suffit pour des tests très simples mais, lorsqu'il faut effectuer un grand nombre de tests, un procédé plus facilement automatisable est nécessaire.

Pour laisser le soin à l'ordinateur (et, plus précisément, au code de test) de vérifier la réussie des cas de tests, on doit prévoir à l'avance non seulement l'appel de la méthode mais aussi le résultat attendu (*expected* en anglais). Si on réalise la liste de tous les appels à effectuer et de toutes les réponses attendues, le code de testing automatisé peut vérifier lui-même si tout se passe bien et afficher le résultat (soit « Tous les tests réussis », soit « Les tests suivants ont échoué : … »).

Les deux méthodes AssertBool et TestValidLogin présentées ci-dessus sont en quelque sorte une introduction à cette structuration des tests (plus à ce sujet dans les laboratoires suivants).

Maintenant que vous disposez d'une méthode pour tester la validité d'un login, assurez-vous que le setter `SetLogin` ne modifie le login en place que s'il s'agit d'un login valide (dans le cas contraire, aucune modification ne sera effectuée).

Testez le setter dans la méthode `Main`.

Étape 3 : une propriété Login

Le procédé qui consiste à associer à un attribut privé `attribut` un getter appelé `GetAttribut()` et un setter appelé `SetAttribut(valeur)` reste utilisable en C#. Cependant, les conventions propres à ce langage reposent sur un autre mécanisme pour protéger l'accès aux données privées (c'est-à-dire pour l'encapsulation) : l'utilisation de propriétés.

On peut voir les propriétés comme un mécanisme permettant d'utiliser une donnée privée comme si elle était publique : la syntaxe utilisée est celle qui correspond à un attribut public, mais il faut bien garder en tête que derrière cette écriture se cache potentiellement un code plus complexe.

Ajoutez le code suivant dans la classe `User`.

```
public string Login
{
  get
  {
    return login;
  }
  set
  {
    if (ForumUtils.ValidLogin(value))
      login = value;
  }
}
```

Ce code crée une propriété appelée `Login`. La plupart du temps, quand on crée une propriété associée à un attribut privé, on utilise le même nom avec une majuscule (ce qui est cohérent avec la convention consistant à mettre une majuscule aux noms de méthodes).

La définition de cette propriété vous permet d'utiliser les syntaxes suivantes partout (pas seulement dans la classe).

```
Console.WriteLine(user.Login);
user.Login = "Mon nouveau login";
```

Lors d'une lecture de la propriété (comme la ligne `Console.WriteLine` ci-dessus), on exécutera le code associé à `get`. Lors d'une écriture de la propriété (comme l'affectation ci-dessus), on utilisera le code associé à `set`. Dans cette seconde partie, l'identificateur `value` est utilisé pour représenter la valeur qu'on tente d'attribuer à la propriété, comme s'il s'agissait d'un appel `SetAttribut(value)`.

Pour bien comprendre comment les propriétés fonctionnent, ajoutez la ligne

```
Console.WriteLine("GET LOGIN");
```

dans la partie `get` du code (avant le `return`, hein !) et la ligne

```
Console.WriteLine("SET LOGIN");
```

dans la partie `set`.

Si, dans la méthode Main, vous ajoutez les lignes suivantes, quels seront les messages envoyés dans la console ? (Donnez une réponse avant de faire le test.)

```
User m = new User ("Cunégonde", "motdepasse");
string nom = m.Login;
m.Login = "Hubert";
m.Login += "ine";
```

Fais le get (pour prendre le nom) ensuite fais le set (mais n'ajoute que le "ine" grâce au +=

Vous pouvez désormais supprimer le code des méthodes `GetLogin` et `SetLogin`. Notez que l'utilisation de propriétés ne contribue pas à rendre le code plus efficace, ni même plus lisible (en fait, on pourrait arguer que le code est moins lisible vu que ce qui ressemble à une simple affectation ou à une lecture d'attribut peut cacher n'importe quel code). Cependant, c'est une convention utilisée dans les programmes en C#, y compris dans la définition de types prédéfinis et dans certains outils qui reposent sur du code C# (comme des liens automatiques entre des bases de données et des propriétés).

Étape 4 : une propriété de plus

Ajoutez à votre classe une nouvelle propriété `JoinDate` qui permettra de manipuler (en lecture et en écriture) la date d'inscription au format numérique AAAAMMJJ.

Dans ce cas-ci, il ne s'agira donc pas juste de lire/écrire le contenu de l'attribut privé ; il faudra effectuer les transformations nécessaires.

Faites une recherche sur Google selon les mots-clefs « C# DateTime » pour trouver de la documentation au sujet de ce type vous permettant de trouver une méthode qui donne directement la date au format demandé (indice : utilisez des propriétés de `DateTime`).

Note. Si vous utilisez de grands nombres tels que 10000, souvenez-vous que vous pouvez les écrire `10_000` si vous préférez cette syntaxe.

Dans la méthode `ToString`, utilisez les deux propriétés créées plutôt que d'accéder directement aux attributs privés.

Ajoutez également un constructeur permettant de créer un membre en donnant sa date d'inscription au format AAAAMMJJ (sous la forme d'un entier donc).

Étape 5 : propriétés partielles

Certains attributs privés doivent être accessibles seulement en lecture, ou seulement en écriture. C'est par exemple le cas de `postCount` : on désire permettre au « monde extérieur » (à la classe) de lire sa valeur, mais pas de la modifier.

On peut créer une propriété qui ne possède qu'une des deux parties (uniquement `get` ou uniquement `set`). Ajoutez le code suivant dans la classe User.

```
public int PostCount
{
   get
   {
      return postCount;
   }
}
```

Si, dans le code de la méthode `Main`, vous joutez une ligne similaire à

```
m.PostCount++;
```

Visual Studio la souligne immédiatement en rouge. Placez votre curseur sur la ligne ondulée rouge et lisez le message d'erreur indiqué.

Si vous ajoutez une partie `set`, le message d'erreur disparaît.

Ajoutez une propriété `Password` qui permet uniquement de modifier le mot de passe, pas d'obtenir celui qui est stocké. En plus de cela, ajoutez une méthode `ValidatePassword(password)` qui indique par un booléen si le mot de passe donné correspond bien à celui qui est encodé.

À nouveau, ajoutez la ligne suivante dans la méthode Main et lisez le message d'erreur produit par l'IDE.

```
Console.WriteLine(m.Password);
```

Étape 6 : un mot de passe haché et salé

C'est généralement très imprudent de stocker dans une base de données des mots de passe « en clair » (c'est-à-dire tels qu'ils sont entrés par l'utilisateur) : en cas de piratage, cette base de données donnerait accès aux mots de passe de tous les utilisateurs !

Au lieu de cela, on stocke généralement une version codée du mot de passe, en utilisant un code qui permet tout de même de distinguer des mots de passe différents. En effet, représenter les mots de passe par leur première lettre serait par exemple une mauvaise idée, car dans ce cas-là, on ne serait plus capable de faire la distinction entre des mots de passe tels que « Anémone » et « Ampoule ». En pratique, il existe plusieurs méthodes de codage prédéfinies (comme par exemple MD5 ou SHA).

Dans le cadre de cet exercice, on utilisera un codage plus simple consistant à faire la somme des codes UTF-8 des caractères du mot de passe puis à calculer le modulo 997 de cette somme.

Dans un premier temps, créez une fonction utilitaire `Encode (string password)` qui encode un mot de passe donné. Notez que la boucle `foreach` peut aussi être utilisée pour passer en revue les uns après les autres les caractères d'une chaîne de caractères, et qu'un caractère peut être automatiquement converti en son code s'il est traité comme un entier.

Modifiez ensuite la classe `User` pour qu'elle ne mémorise plus le mot de passe mais son code. Pour ce faire, commencez par faire en sorte que la propriété `Password` reçoive un mot de passe sous la forme d'une chaîne de caractères mais ne stocke que son code. Dans ce cas-ci, derrière une propriété de type `string`, on aura une valeur entière (un attribut privé de type entier). N'oubliez pas de mettre à jour les constructeurs, la méthode `ValidatePassword` et la méthode `ToString`.

Note. C# possède des méthodes prédéfinies pour coder en SHA_512. Ceux qui sont intéressés peuvent rechercher des informations à ce sujet sur la documentation en ligne et modifier leur code pour les utiliser.

Exercice 2 : Messages du forum

Étape 1 : la classe Post

Créez une classe `Post` conservant les informations suivantes.

- `author` : l'utilisateur qui a écrit le message
- `contents` : le contenu (textuel) du message
- `date` : la date et l'heure du jour où le message a été posté
- `likedBy` : un tableau d'utilisateurs ayant « liké » le message (initialement vide)

Considérez les deux options suivantes et pesez le pour et le contre de chacune d'entre elles.

- Option 1 : mémoriser dans un attribut le nombre d'utilisateurs ayant « liké » le message.
- Option 2 : utiliser des « null » dans les cellules inutilisées de `likedBy`.

Créez un constructeur standard recevant la référence de l'auteur, le contenu et la date, ainsi qu'une méthode `ToString` adéquate. Ajoutez également un constructeur ne fournissant que l'auteur et le contenu du message. Dans tous les cas, n'oubliez pas de faire appel à la méthode `AddPost` de l'utilisateur.

Étape 2 : incréments de taille

Le tableau `likedBy` reprendra les références des membres qui auront « liké » le message en question. Dans ce genre de cas, il est plutôt difficile de savoir à l'avance quelle taille donner au tableau.

On pourrait théoriquement réserver un emplacement mémoire correspondant exactement au nombre de « likes » reçu par le message. En cas de nouveau « like », il faudrait alors réserver un nouvel emplacement, avec une cellule de plus, puis recopier les anciens « likes » dans le nouveau tableau et y ajouter le dernier « like » en date. Si ces copies de tableaux sont effectuées trop souvent, cela peut réduire fortement l'efficacité du code.

Dans ce genre de cas, on peut employer une technique basée sur un **incrément de tableaux**. En deux mots, plutôt que de recopier le tableau et d'ajouter 1 seule cellule quand c'est nécessaire, on recopie le tableau et on ajoute un certain nombre de cellules, assez petit pour que cela ne constitue pas un gaspillage de mémoire et assez grand pour qu'on n'ait pas à recopier le tableau trop souvent. Le nombre en question est appelé l'incrément. À tout moment, on conserve un tableau dont la taille est un multiple de l'incrément (ce qui sur certains systèmes rend les accès mémoire plus rapides).

Ajoutez une constante de classe `INC_LIKEDBY_SIZE` valant par défaut 3 (c'est une valeur sans doute plus petite que ce qu'on utiliserait en pratique, mais elle permettra d'établir plus facilement des tests).

À la création d'un message, le tableau `likedBy` sera initialisé à une taille de `INC_LIKEDBY_SIZE` cellules contenant toutes `null`.

Codez une méthode `AddLike(User user)` qui ajoute un membre à la liste de ceux qui ont « liké » le message. Si le tableau n'a pas assez de place[1], il faudra réserver un nouveau tableau (avec `INC_LIKEDBY_SIZE` cellules de plus que le précédent), recopier les anciennes valeurs et ajouter le nouveau membre.

Modifiez `ToString` pour qu'il renvoie la liste des logins des membres qui ont « liké » un message pour faciliter vos tests. Est-ce une bonne idée d'utiliser le type `string` à l'intérieur pour la construction de la chaîne de caractères à renvoyer ?

Ajoutez également une méthode `RemoveLike(User user)` qui supprime l'utilisateur donné de la liste de ceux qui ont « liké » le message (s'il est présent !).

Étape 3 : paramètres variables

Surchargez `AddLike` en ajoutant une seconde version qui reçoit un tableau de membres plutôt qu'un seul membre.

Testez cette nouvelle définition via une instruction du genre

```
post.AddLike(new User[] {u1, u2, u3, u4});
```

où `post` est un message et `u1`, `u2`, `u3` et `u4` des utilisateurs.

Dans la définition de la seconde version de `AddLike`, ajoutez le mot-clef `params` juste avant la déclaration du paramètre formel `User [] userList`.

Ce mot-clef permet d'appeler la méthode en citant les paramètres un par un plutôt qu'en les rassemblant dans un tableau.

```
post.AddLike(u1, u2, u3, u4);
```

Note. C'est l'équivalent des syntaxes `addLike(User… userList)` en Java et `addLike (…userList)` en Javascript.

Dans un premier temps, utilisez un simple appel à `AddLike` pour chaque utilisateur à ajouter (note : utilisez un `foreach`).

Cette implémentation est très inefficace, tout particulièrement dans le cas où on ajoute un grand nombre d'utilisateurs. Voyez-vous pourquoi ? Modifiez le code en conséquence.

Étape 4 : statistiques d'espaces et de chiffres

Revenez à la classe `User` et ajoutez-lui les attributs suivants dont le but sera de conserver des statistiques sur les messages écrits par ce membre.

- `spacesPosted` : entier décrivant le nombre d'espaces que le membre a utilisés dans ses messages

[1] Pour information, dans certaines versions de cette approche, on « étend » le tableau (en fait, on crée une copie plus grande) plus tôt, dès qu'on se rend compte qu'il est rempli à 90% par exemple. Dans un programme à plusieurs processus, ça permet de continuer à travailler pendant que le tableau se copie plutôt que d'interrompre l'exécution.

- `digitsPosted` : entier décrivant le nombre de chiffres que le membre a utilisés dans ses messages

Initialement, ces attributs seront mis à 0.

Dans la classe utilitaire `ForumUtils`, ajoutez des méthodes `SpaceCount (string txt)` et `DigitCount (string txt)` qui renvoient respectivement le nombre d'espaces et le nombre de chiffres que contient un texte.

Dans la classe `User`, modifiez `AddPost` pour qu'il reçoive le message comme paramètre et mette également à jour les deux nouveaux attributs `spacesPosted` et `digitsPosted`.

Étape 5 : paramètres de sortie

On constate clairement que les codes des méthodes `SpaceCount` et `DigitCount` sont similaires et que faire un appel à l'une de ces méthodes puis à l'autre n'est pas efficace vu qu'on va parcourir deux fois un message potentiellement long. Il faudrait donc plutôt obtenir les deux résultats (nombre d'espaces et nombre de chiffres) en un seul passage du tableau.

Ajoutez une troisième méthode de classe à `ForumUtils` en utilisant l'en-tête suivante.

```
public static void Count (string txt, out int spaceCount, out int
digitCount)
```

Le mot-clef "`out`" indique que les paramètres `spaceCount` et `digitCount` sont des sorties de la méthode qui ne devront pas être renvoyées explicitement via une instruction `return`. Concrètement, cela signifie qu'à la fin de l'exécution de la méthode, leur valeur sera recopiée dans les variables citées lors de l'appel.

Complétez le code de la méthode `Count`. Notez que vous ne devez plus y déclarer les variables `spaceCount` et `digitCount` (mais vous devez toujours les initialiser) !

Du côté de `AddPost(Post post)`, vous pouvez désormais utiliser les lignes

```
int spaceCount;
int digitCount;
ForumUtils.Count(contenu du message, out spaceCount, out digitCount);
```

qui vont garnir les variables locales `spaceCount` et `digitCount` avec les valeurs calculées par la méthode `Count`. Il suffira ensuite d'ajouter ces valeurs aux attributs du membre.

Vous pouvez utiliser les lignes suivantes pour tester votre code (en vous arrangeant pour que les valeurs des attributs soient affichées au lieu des commentaires).

```
User u = new User("Mélusine", "12345");
// 0 post, 0 space, 0 digit
Post p1 = new Post(u, "Un deux trois");
// 1 post, 2 spaces, 0 digit
Post p2 = new Post(u, "456");
// 2 posts, 2 spaces, 3 digits
```

```
Post p3 = new Post(u, "7 8 neuf !");
// 3 posts, 5 spaces, 5 digits
```

Étape 6 : paramètres par référence

On peut encore considérer une autre approche : faire en sorte que la fonction utilitaire mette automatiquement à jour les attributs.

Copiez/collez le code de la méthode `Count` et modifiez-en l'en-tête comme indiqué ci-dessous.

```
public static void CountAndUpdate (string txt, ref int spaceCount, ref
int digitCount)
```

Cette fois-ci, le mot-clef "`ref`" indique que les arguments sont passés par référence ; autrement dit, la méthode `CountAndUpdate` va directement travailler sur les emplacements mémoires des variables passées.

Cela signifie qu'il faudra légèrement modifier le code de `CountAndUpdate` pour que cela fonctionne (à vous de voir quoi changer).

Du côté de `AddPost`, l'appel s'écrira désormais

```
ForumUtils.CountAndUpdate(contenu du message, ref spacesPosted, ref
digitsPosted);
```

et utilisera directement les attributs du membre. (Notez que, même si ces attributs sont privés, le fait de les passer par référence expose leur emplacement mémoire et permet donc à `CountAndUpdate` de les modifier directement).

Utilisez à nouveau le code proposé dans l'étape précédente pour tester.