MS860

Programmation C# avec .NET

Livret d'ateliers

- Module 1 – Introduction à C# et au .Net Framework

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer un programme en C# ;
- compiler et exécuter un programme en C# ;
- utiliser le débogueur Visual Studio;

Exercice 1
Création d'un programme simple en C#

Dans cet exercice, vous allez utiliser **Visual Studio** pour écrire un programme simple en C#. Le programme vous demandera d'indiquer votre nom et vous accueillera ensuite avec celui-ci.

- **Pour créer une nouvelle application console C#**

1. Démarrez **Visual Studio** et dans la fenêtre qui s'ouvre, choisissez **Créer un projet**.

2. Dans la liste des modèles de projet proposés, retrouvez **Application Console C#** (attention à ne pas choisir *Application Console .NET Framework*). Vous pouvez vous aider du champ de recherche en tapant **console**. Une fois le type de projet sélectionné, cliquez sur **Suivant**.

3. Choisissez le dossier de votre choix et renommez la solution en **GreetingsSln** et le projet en **Greetings**. Cliquez sur **Suivant**.

4. Notez la possibilité de choisir sa version de framework, et laissez la valeur par défaut (à savoir .NET 6). Cliquez sur **Créer**.

5. Ouvrez le fichier **Extrait.txt** du dossier **Ateliers/Mod01/Starter** et copiez son contenu pour le coller dans le fichier **Program.cs** à la place du code généré automatiquement par **Visual Studio**.

Note : nous verrons plus tard que le code initial était tout à fait légitime, étant une version simplifiée du code que nous avons collé.

6. Enregistrez votre travail.

7. Testez votre programme sans lancer le débogage (Menu **Déboguer**, **Exécuter sans débogage** ou **Ctrl+F5**). Notez la fenêtre de commande qui s'ouvre et dans laquelle s'affiche le message *Hello, World !*. Comme indiqué, tapez une touche pour terminer l'exécution.

Pour écrire des instructions d'invite et d'accueil de l'utilisateur

1. Dans le fichier *Program.*cs, localisez la méthode **Main**, et après l'accolade ouvrante, insérez la ligne suivante en lieu et place de la ligne *Console.WriteLine("Hello, World !") ;* :

```
string myName;
```

2. Écrivez une instruction pour inviter les utilisateurs à indiquer leur nom à l'aide de la méthode *Console.WriteLine()*.

3. Écrivez une autre instruction pour lire la réponse de l'utilisateur à partir du clavier et l'assigner
à la chaîne *myName*, à l'aide de la méthode *Console.ReadLine()*.

4. Ajoutez une autre instruction pour imprimer « Bonjour *myName* » à l'écran (où *myName* est
le nom saisi par l'utilisateur). Pour cela, vous pouvez utilisez le symbole mathématique + pour « ajouter » les deux morceaux de chaînes de caractère (la chaîne fixe « Bonjour » et la variable *myName*).

5. Une fois cette opération terminée, la méthode **Main** devrait ressembler au code suivant :

```
public static void Main(string[ ] args)
{
  string myName;
  Console.WriteLine("Entrez votre nom");
  myName = Console.ReadLine( );
  Console.WriteLine("Bonjour " + myName);
}
```

6. Enregistrez votre travail (Ctrl+S).

Note : il se peut que le code précédent expose un avertissement de conversion sur *Console.ReadLine()*. N'en tenez pas compte.

- **Exécution avec ou sans débogage**

1. Comme précédement, lancez le programme sans débogage (**Ctrl+F5**), tapez votre nom lorsque demandé et tapez Entrée pour constater le message personnalisé. Tapez une nouvelle touhe pour fermer la fenêtre.

2. Relancez le programme, cette fois, en intégrant le débogage (Menu **Déboguer**, **Démarrer le débogage** ou **F5**). Saisissez votre nom et tapez Entrée. Si la fenêtre se ferme automatiquement, veuillez modifiez l'option suivante de Visual Studio : Menu Outils -> Options -> Débogage et décochez l'option Fermer automatiquement la console à l'arrêt du débogage (elle se situe dans le bas de la liste déroulante).

Ne fermez pas Visual Studio, nous allons nous en resservir dans l'exercice suivant.

Exercice 2
Utilisation du débogueur

Dans cet exercice, vous allez utiliser le débogueur Visual Studio Code pour parcourir votre programme pas à pas et examiner la valeur d'une variable.

- **Pour définir un point d'arrêt et commencer le débogage à l'aide de Visual Studio Code**

1. Repérez la ligne contenant la première occurrence de **Console.WriteLine** dans la classe **Program**.

2. Positionnez un point d'arrêt.
Pusieurs solutions sont possibles pour positonner un point d'arrêt. Vous pouvez aller dans le menu **Déboguer** et choisir **Basculer le point d'arrêt** (F9) ou bien cliquer à gauche de la ligne dans l'éditeur (juste avant le numéro de ligne, un point gris apparaît au survol de la souris et devient rouge une fois le point d'arrêt positionné).

Note : si la numérotation des lignes n'est pas activée, vous pouvez modifier l'option dans Menu **Outils** -> **Options** -> **Editeur de Texte** -> **C#**

3. Dans le menu **Déboguer**, cliquez sur **Démarrer le débogage** (F5).
L'exécution du programme commence, la console de débogage s'affiche, et le programme s'interrompt au point d'arrêt (la ligne sélectionnée est à présent surlignée en jaune).

- **Pour surveiller la valeur d'une variable**

1. Notez que Visual Studio affiche les différents panneaux de débogage : Variables locales,, Espion 1, Pile des appels, outils de diagnostic…

2. Dans le panneau **Espion 1**, ajoutez la variable *myName* à la liste des variables surveillées (en cliquant sur le texte *Ajouter un élément à espionner*).

3. La variable *myName* s'affiche dans le panneau **Espion 1** avec la valeur *null*.

- **Pour parcourir le code pas à pas**

1. Dans le menu **Déboguer**, cliquez sur **Pas à pas principal** (F10 ou via la barre d'outils de Visual Studio) pour exécuter la première instruction **Console.WriteLine**. Constatez que le texte de demande s'affiche dans le **Terminal**.

2. Passez à la ligne suivante contenant l'instruction **Console.ReadLine** en appuyant sur F10.

3. Retournez dans le **Terminal** et tapez votre nom, puis appuyez sur la touche **ENTRÉE**.
La valeur de *myName* dans la fenêtre Espion correspond à présent à votre nom.

4. Amenez le curseur de votre souris au-dessus de la variable *myName* dans le code et constatez l'info-bulle affichant le contenu de la variable. Notez qu'il est ici possible de modifier la valeur ou bien de le faire via le panneau de débogage (espions, variables locales).

5. Vous pouvez à présent continuer l'exécution en pas à pas, ou tout simplement finir l'exécution d'un bloc, en choisissant **Continuer** (F5).

- Module 2 – Structures de programmation C#

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- mettre en œuvre des variables et des structures de programmation en C# ;
- utiliser des tableaux en C# ;

Exercice 1
Reconnaître des erreurs de syntaxe en C#

Dans cet exercice, vous allez devoir corriger des erreurs de syntaxe en C#.

- **Trouver les erreurs de syntaxe dans les propositions suivantes :**

  - if number % 2 == 0 …
  - if (percent < 0) || (percent > 100) …
  - if (minute == 60) ;
    minute = 0 ;
  - for (int i=0 , i < 10, i++)
    Console.WriteLine(i) ;
  - int i = 0;
    while(i < 10)
    Console.WriteLine(i);
  - for (int i=0; i >= 10; i++)
    Console.WriteLine(i);
  - do
    …
    string line = Console.ReadLine();
    guess = int.Parse(line);
    while (guess != answer);
  - int[] array;
    array = {0, 2, 4, 6};
  - int[] array;
    Console.WriteLine(array[0]);
  - int[] array = new int[3];
    Console.WriteLine(array[3]);
  - int[] array = new int[];
  - int[] array = new int[3]{0, 1, 2, 3};

Exercice 2
Le jeu du plus ou moins

Dans cet exercice, vous allez écrire un programme ludique, le jeu du plus ou moins. Le but est
de générer une valeur aléatoire entre 0 et 100 et de proposer au joueur de trouver cette valeur
par essais successifs. Pour tester votre programme, vous pouvez soit déboguer dans Visual Studio Code, soit lancer en ligne de commande *dotnet run*.

- **Mise en œuvre de la structure de base du jeu**

1. En réutilisant les indications de l'atelier précédent, créez un nouveau projet de type **Console**, nommé **JeuPoM** dans une solution **JeuPoMSln**, et enregistré sous **.\Ateliers\Mod02\Starter**.

2. Copiez le contenu du fichier **Extrait.txt** du dossier **.\Ateliers\Mod02\Starter** à la place du code contenu dans le fichier **Program.cs** ouvert dans **Visual Studio**.

3. Dans la méthode *static void Main*, déclarez les différentes variables nécessaires : *valeurSecrete*
et *valeurSaisie* (de type entier), *reponse* (de type chaîne de caractères) et une variable *rnd* de type **Random** avec la syntaxe suivante :

```
Random rnd = new Random() ;
```

4. Initialisez la valeur secrète à l'aide la méthode **Next** de la variable *rnd*, en précisant une valeur maximum de 100 à l'aide du paramètre de cette méthode.

5. Interrogez l'utilisateur à l'aide la méthode **Console.ReadLine** afin de récupérer la valeur saisie. Pour cela, vous devrez utiliser la méthode **int.Parse** qui permet de convertir la chaîne de caractères saisie en entier. Nous supposerons pour l'instant que l'utilisateur ne commet pas de fautes
de frappes.

```
int unEntier = int.Parse(uneChaineDeCaractère) ;
```

6. Effectuez les comparaisons nécessaires entre la valeur saisie et la valeur secrète afin d'indiquer
au joueur par le biais de messages s'il a saisi une valeur trop grande, trop petite ou correcte.

7. Testez le programme.

- **Mise en œuvre de structures de boucles**

1. Afin que le joueur retente sa chance pour trouver la bonne valeur, utilisez une structure de boucle pour encadrer à la fois la saisie d'une valeur et les tests par rapport à la valeur secrète.
La condition de sortie de la boucle portera sur l'égalité des deux valeurs.

2. Après avoir indiqué au joueur qu'il a trouvé la bonne valeur (à savoir après la boucle précédente), interrogez à nouveau l'utilisateur sur sa volonté de rejouer une nouvelle partie. La variable *reponse* nous permettra de stocker la réponse fournie par le joueur.

3. A l'aide d'une nouvelle structure de boucle, encadrez ce qui constitue une partie (initialisation
de la valeur secrète, répétition de saisie utilisateur, annonce de la victoire et demande pour

une nouvelle partie).
La condition de sortie portera sur le contenu de la variable *reponse*, et pourra permettre de tester
si l'utilisateur a répondu *oui* à la question qui lui est posée.

Remarque : l'utilisation de boucles de type do…while peut permettre d'alléger le code et fournir
une structure comme suit.

```
// déclaration des variables

do{
  // initialisation de la valeur secrète
  do{
    // récupération de la saisie utilisateur
    // tests entre valeur secrète et valeur saisie ( cas < et >)
  }while(valeurSecrete !=valeurSaisie) ;
  // annonce de la victoire
```

4. Testez le programme.

- **Décompte du nombre de tentatives**

1. Déclarez deux nouvelles variables de type entier, *nbTentative* qui permettra de compter le nombre de tentatives pour trouver la bonne valeur, et *meilleurScore* qui stockera le plus petit nombre
de tentative pour l'ensemble des parties dans une session de jeu (i.e. pendant une exécution du programme).
La variable *meilleurScore* sera initialisée arbitrairement à 50, ce qui assure au joueur de fortes probabilités de battre ce meilleur score lors de sa première partie.

2. Incrémentez la variable *nbTentative* à chaque nouvelle saisie du joueur.

3. Après avoir indiqué au joueur qu'il a trouvé la bonne valeur, comparez le nombre de tentatives
en court avec le meilleur score afin de mettre à jour celui-ci en cas d'amélioration.

4. Avant de proposer au joueur de refaire une nouvelle partie, indiquez-lui s'il a ou non réalisé
le meilleur score ainsi que le nombre de coups qu'il aura mis pour trouver la bonne valeur.

5. Testez le programme.

- **Gestion d'un historique des parties**

1. Déclarez deux tableaux d'entier (*historiqueValeur* et *historiqueTentative*) de même taille (20 éléments) qui serviront à stocker l'historique des parties pendant une session de jeu.

2. Déclarez un entier *nbParties* afin de stocker le nombre de parties jouées pendant la session.
Cette variable permettra également d'accéder aux cellules correspondantes des 2 tableaux précédemment créés.

3. Incrémentez le nombre de parties après avoir initialisé la valeur secrète et stockez cette valeur dans la cellule correspondante du tableau d'historique de valeurs.

4. Stockez le nombre de tentative dans le tableau correspondant lorsqu'une partie est terminée.

5. Une fois que le joueur a décidé de quitter le programme, affichez l'historique des parties sous
la forme « Partie N°… , valeur secrète=… , trouvé en … coup(s). ».
Les informations nécessaires se trouvant dans deux tableaux différents, il n'est toutefois pas nécessaire de réaliser deux structures de boucle.

6. Testez le programme.

• Module 3 – Déclaration et appel de méthodes

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer des méthodes en C# ;
- utiliser des méthodes en C# ;

Exercice 1
Modularisation d'un code existant

Dans cet exercice, vous allez mettre en œuvre des méthodes dans le programme *Jeu* introduit
dans le module précédent. Plus précisément, il sera question d'extraire du code principal, les interactions avec le joueur ainsi que l'affichage de l'historique des parties.

- **Modularisation de l'affichage d'historique**

1. Ouvrez la solution .**\Ateliers\Mod03\Starter\JeuPoMSln\JeuPoM.sln** dans Visual Studio.

2. Localisez le commentaire **TODO : Exercice 1.1**.

3. Mettez en place une nouvelle méthode A*ffichеHistorique* ne retournant rien (**void**) et acceptant
en paramètre les informations suivantes : *compteur* (entier), *tabValeur* (tableau d'entier) et *tabCoup* (tableau d'entier). Cette méthode sera également **static** comme l'est la méthode *Main*, l'utilisation de ce mot clé étant justifiée ultérieurement.

4. Localisez le code initial d'affichage d'historique (commentaire **TODO : Exercice 1.2**) et après l'avoir déplacé dans le corps de la nouvelle méthode, remplacez-le par l'appel d'A*ffichеHistorique*
en lui passant en paramètre les bonnes données.

5. Corrigez le code de la fonction A*ffichеHistorique* collé à l'étape précédente, en tenant compte
des nouveaux noms de variables.

6. Testez le programme.

- **Modularisation des interactions utilisateurs**

1. Localisez le commentaire **TODO : Exercice 1.3**. Mettez en œuvre deux nouvelles fonctions acceptant une chaîne de caractères *message* en paramètre et retournant respectivement une chaîne de caractères (fonction *GetString*) et un entier (fonction G*etEntier*).

Ces deux fonctions auront pour but d'afficher la chaîne *message* sur la sortie standard et d'interroger l'utilisateur pour récupérer une valeur saisie au clavier. Il s'agit donc de remplacer les lignes d'interaction utilisateur pointées par les commentaires **TODO : Exercice 1.4** et **TODO : Exercice 1.5**.

La code des deux fonctions est relativement similaire et peut se résumer ainsi :

Fonction (paramètre Chaîne)
Affichcer Chaîne à l'écran
Récupérer la saisie utilisateur
Renvoyer une valeur

Dans le cas de *GetString*, le retour sera exactement la saisie utilisateur.

Dans le cas de la fonction G*etEntier*, il faut retourner le résultat de la conversion de la chaîne saisie en entier. Pour cela, nous allons utiliser la fonction *int.TryParse* dont la signature est la suivante :

```
bool TryParse (string chaineEntree, out int entierSortie)
```

Cette fonction retourne un booléen en fonction de la réussite ou non de la conversion de la chaîne entrée et retourne la valeur convertie en entier dans la variable *entierSortie*. Nous considérerons
que si la chaîne n'est pas un entier, la valeur retournée par G*etEntier* sera égale a -1.

Comme pour la fonction A*ffichageHistorique*, les deux méthodes seront marquées *static*.

2. Localisez le commentaire **TODO : Exercice 1.4** et mettez en œuvre la fonction G*etEntier* de façon
à stocker le résultat de la fonction dans la variable *valeurSaisie*.

3. Localisez le commentaire **TODO : Exercice 1.5** et mettez en œuvre la fonction *GetString* de façon
à stocker le résultat de la fonction dans la variable *reponse*.

4. Testez le programme.

- Module 4 – Gestion d'exceptions

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- mettre en évidence les problèmes de dépassement de capacité;
- déclencher et capturer des exceptions ;

Exercice 1
Les dépassements de capacités en C#

Dans cet exercice, vous allez découvrir la problématique du dépassement de capacité et voir comment mettre en œuvre les sections **checked** pour contrôler ces problèmes.

- **Les dépassements de capacité en C#**

1. Ouvrez la solution *DepCap.sln* dans le dossier .**\Ateliers\Mod04\Starter\DepCap**.

2. Déclarez une variable entière *number* dans la méthode **Main** et affectez-lui la valeur maximum du type *int* (**int.MaxValue**).

3. A l'aide de **Console.WriteLine**, affichez sur la console la valeur de *number* puis la valeur de ++*number*.

4. Testez le programme. Que constatez-vous ?

Explications : Nous venons de mettre en évidence le problème de dépassement de capacité en C#. Par défaut, lorsqu'une variable atteint la valeur maximum de son type, une augmentation supplémentaire ne déclenche pas d'exception, et au contraire, la variable fait « le tour du compteur » se voyant affectée de la plus petite valeur du type.

Ce comportement par défaut peut être modifié de trois façons différentes : utilisation d'options
de compilation, instruction *checked* ou expression *checked*. Dans tous les cas, un tel dépassement
de capacité arithmétique engendrera une exception du type **OverflowException**.

5. Pour utiliser l'instruction *checked*, placez votre code précédent dans un bloc $\mathrm{checked}\{\ldots\}$.

6. Testez le programme et constatez la levée d'exception.

Remarque : Il aurait également été possible de mettre en œuvre l'expression *checked* sous la forme suivante : *Console.WriteLine(checked(++number)) ;*

7. Ajoutez au code précédent une structure *try/catch* afin de capturer cette exception et assurer
la continuité du programme en affichant un message en cas d'erreur.

8. Testez le programme et assurez-vous qu'aucune exception n'est déclenchée et que votre message s'affiche à la place.

Exercice 2
Levée d'exception

Dans cet exercice, vous allez apprendre à déclencher une exception dans le corps d'une méthode
de sorte que le code appelant sache que quelque chose ne s'est pas exécuté correctement. En particulier, vous allez utiliser cette approche dans le *Jeu* pour signaler à l'utilisateur une saisie
non valide.

- **Mise en œuvre de la levée d'exception**

1. Ouvrez la solution *JeuPoMSln.sln* du dossier **.\Ateliers\Mod04\Starter\JeuPoMSln** dans Visual Studio.

2. Localisez le commentaire **TODO : Exercice 2**.

3. Remplacez la logique actuelle de sorte qu'une exception soit levée lorsque la méthode *int.TryParse* ne peut effectuer la conversion de chaîne. Le message de cette exception sera « La valeur saisie n'est pas valide. ».

4. Enregistrez votre travail.

Exercice 3
Capture d'exception

Dans cet exercice, vous allez utiliser l'instruction **try/catch** pour capturer une exception provenant d'une sous-méthode et assurer la continuité du programme principal. En particulier, suite
à la modification apportée dans l'exercice précédent, vous allez mettre en œuvre une structure
de gestion d'erreurs lors de l'utilisation de la fonction *GetEntier*.

- **Capture d'exception**

1. Localisez le commentaire **TODO : Exercice 3**.

2. Mettez en œuvre l'instruction *try/catch* de façon à capturer l'exception que peut déclencher
la méthode *GetEntier*.

3. Placez dans le bloc *catch* le code nécessaire à l'affichage du message d'erreur, et faites en sorte que l'itération en court soit arrêtée sans toutefois arrêter la boucle.

4. Déplacez le code d'incrémentation du nombre de tentatives, de sorte qu'une saisie erronée
de l'utilisateur compte comme un coup pour rien.

5. Testez le programme et vérifiez qu'une erreur de saisie ne bloque pas le programme.

- Module 5 – Lire et écrire dans des fichiers

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- lire le contenu d'un fichier texte;
- écrire dans un fichier texte ;

Exercice 1
Persistance d'informations dans un fichier

Dans cet exercice, vous allez mettre en œuvre une solution de persistance de données pour le programme de Jeu. Le meilleur score sera stocké pour être lu à l'exécution suivante ; l'historique pourra être proposé sous forme de fichier.

- **Génération d'un fichier d'historique**

1. Ouvrir la solution *JeuPoM.sln* du dossier **.\Ateliers\Mod05\Starter\JeuPoMSln** dans Visual Studio.

2. Dans le fichier *Program.cs*, ajoutez l'instruction *using* nécessaire à l'utilisation des classes de **System.IO**.

3. Localisez le commentaire **TODO : Exercice 1.1**.

4. Ecrivez une surcharge à la méthode A*ffichеHistorique*, acceptant un quatrième paramètre *nomFichier* de type chaîne de caractères, en dupliquant la méthode existante.

5. Au début de la nouvelle méthode, ajoutez les déclarations et instanciations des objets *FileStream* et *StreamWriter* nécessaires à l'écriture dans le nouveau fichier. Nous prendrons en particulier
les paramètres **FileMode.Create** et **FileAccess.Write** pour l'objet *FileStream*.

6. Modifiez le reste du code de la méthode en remplaçant l'objet *Console* par l'objet *StreamWriter* créé à l'étape précédente.

7. A la fin de la méthode, appelez les méthodes de fermeture (**Close**) des objets *StreamWriter*
et *FileStream* de sorte à fermer correctement le fichier nouvellement créé.

8. Utilisez une instruction *try/catch* pour éviter tout problème durant cette procédure. Un simple message d'erreur sera retourné à l'utilisateur en cas d'exception.

- **Nouvelle gestion de l'affichage d'historique pour le joueur**

1. Localisez le commentaire **TODO : Exercice 1.2**.

2. Utilisez la fonction *GetString* pour demander à l'utilisateur un nom de fichier pour sauvegarder
son historique. Vous pouvez pour cela réutiliser la variable *reponse*.

3. Testez la valeur saisie par l'utilisateur. En cas de chaîne de caractères vide, il sera alors effectué

un affichage de l'historique sur l'écran. Dans le cas contraire, le nom de fichier spécifié sera utilisé avec la nouvelle méthode A*ffichеHistorique* pour stocker physiquement l'historique.

4. Testez votre programme. Si l'utilisateur ne saisit que le nom de fichier (sans chemin d'accès complet), celui-ci sera stocké dans le dossier d'exécution du programme, dans notre cas **.\Ateliers\Mod05\Starter\JeuPoMSln\JeuPoM\bin\Debug\net6.0**. De plus, l'extension est à charge de l'utilisateur.

Exercice 2
Persistance du meilleur score

Dans cet exercice, vous allez écrire le code nécessaire à la sauvegarde du meilleur score en fin
de session de jeu.

- **Sauvegarde du meilleur score**

1. Localisez le commentaire **TODO : Exercice 2**.

2. Déclarez et initialisez deux variables de type *FileStream* et *StreamWriter* de sorte d'ouvrir un fichier **high.txt** en écriture, avec remplacement du fichier s'il existe déjà.

3. Sauvegardez la valeur du meilleur score dans le fichier.

4. Fermez les objets *StreamWriter* et *FileStream*.

5. Testez le programme.

Exercice 3
Lecture du meilleur score

Dans cet exercice, vous allez lire le fichier précédemment créé, lors du démarrage du programme.

- **Lecture du fichier de sauvegarde en début de session**

1. Localisez le commentaire **TODO : Exercice 3**.

2. Déclarez et initialisez les variables *FileStream* et *StreamReader* nécessaires à la lecture du fichier **high.txt**.

3. A l'aide de l'objet *StreamReader*, lisez la valeur écrite dans le fichier de sauvegarde et stockez-la dans la variable *meilleurScore*. La conversion sera effectuée à l'aide de *int.Parse*.

4. Fermez les objets *StreamReader* et *FileStream*.

5. Dans l'hypothèse où le fichier aurait été modifié, supprimé, ou encore que sa lecture pose problème, encadrez le code précédent dans une structure de gestion d'erreur.
En cas d'erreur, la valeur de la variable *meilleurScore* sera celle par défaut.

6. Testez le programme.

- **Amélioration de la lecture par anticipation des exceptions**

1. Quel problème peut être engendré par le code précédent ?

2. En utilisant des fonctions de test (existence d'un fichier, possibilité de conversion), réécrivez
le code précédent de sorte à éviter une structure de gestion d'exception de *type try/catch*.

3. Testez le programme.

- Module 6 – Création de nouveaux types de données.

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer une classe ;
- utiliser une classe ;
- mettre en œuvre une bibliothèque de classe ;

Exercice 1
Création d'une classe

Dans cet exercice, vous allez créer une nouvelle classe permettant de représenter une partie, à savoir l'ensemble de données *valeur à trouver* et *nombre de tentatives*. Ce nouveau type permettra en particulier de simplifier la gestion de l'historique par l'introduction d'un nouveau tableau unique.

- **Création de la nouvelle classe Partie**

1. Ouvrez la solution du dossier **.\Ateliers\Mod06\Ex1\Starter\JeuPoMSln** dans Visual Studio.

2. Via l'explorateur de solutions, ajouter une nouvelle classe dans un fichier **Partie.cs** .

3. Dans le corps de la classe *Partie*, créez 3 régions nommées **Attributs**, **Constructeurs** et **Méthodes** selon la syntaxe suivante :

```
#region NomDeRégion

#endregion
```

4. Dans la région **Attributs**, créez deux champs publics entiers *valeur* et *tentative*.

5. Dans la région **Constructeurs**, créez deux constructeurs publics pour la classe. Le premier sera sans paramètre, le deuxième acceptera un seul paramètre pour initialiser le membre *valeur*.

6. Dans la région **Méthodes**, créez une méthode publique *Info*, retournant une chaîne de caractères du type « *valeur* trouvé en *n* coup(s) », et n'acceptant aucun paramètre.

7. Enregistrez votre travail.

- **Utilisation de la classe Partie**

1. Localisez le commentaire **TODO : Exercice 1.1** dans le fichier *Program.cs*.

2. Remplacez les déclarations des tableaux *historiqueValeur* et *historiqueTentative* par la déclaration d'un nouveau tableau *historique* de type *Partie*. La taille sera toujours fixée à 20 éléments.

```
Partie[] historique = new Partie[20] ;
```

3. Localisez le commentaire **TODO : Exercice 1.2**.

4. Remplacez l'utilisation du tableau *historiqueValeur* par l'instanciation d'un nouvel objet de type *Partie* qui sera stocké dans le tableau *historique* en position *nbParties*. Il s'agira de la partie en cours.

5. Localisez le commentaire **TODO : Exercice 1.3**.

6. Remplacez l'utilisation du tableau *historiqueTentative* par la mise à jour de la *Partie* en cours
par le biais du tableau *historique*.

7. Localisez le commentaire **TODO : Exercice 1.4**. Constatez les erreurs sur les méthodes *AfficheHistorique*.

8. Corrigez les méthodes A*fficheHistorique*, de sorte qu'elles prennent un tableau de *Parties* en entrée en remplacement des deux tableaux d'entiers précédents. De plus, lors de la mise en forme du message, utilisez la méthode *info* de la classe *Partie* pour construire la chaîne de caractères. Les nouvelles signatures de ces méthodes seront :

```
static void AfficheHistorique(int compteur, Partie[] tableau)
static void AfficheHistorique(int compteur, Partie[] tableau, string nomFichier)
```

9. Corrigez les appels aux fonctions A*fficheHistorique* en tenant compte des modifications précédentes.

10. Testez le programme.

Exercice 2
Mise en œuvre d'une bibliothèque de classes

Dans cet exercice, vous allez créer puis utiliser une bibliothèque de classes. La solution finale comportera donc deux sous-projets, celui de la bibliothèque de classes et celui du projet exécutable. L'utilisation de la classe *Personne* dans le *Jeu*, se fera de la manière suivante : au début
du programme, il sera demandé au joueur s'il veut s'authentifier en précisant son nom et son prénom. S'il accepte, dans ce cas ces informations seront utilisées pour la génération de l'historique.

- **Création d'une bibliothèque de classes**

1. Ouvrir la solution du dossier **.\Ateliers\Mod06\Ex2\Starter\JeuPoMSln** dans Visual Studio.

2. Ajouter un nouveau projet de type **Bibliothèque de classe** au projet, et nommez-la **Info**. NOTE : Effectuez un clic-droit sur le nœud *Solution* 'JeuPoMSln' dans l'Explorateur de Solutions et choisissez **Ajouter**/**Nouveau Projet**. Attention à bien choisir un projet **Bibliothèque de classe** en **C#** et sans aucune autre indication dans son nom.

3. Renommez le fichier *Class1.cs* en *Personne.cs*. Visual Studio devrait vous proposer de renommer également toutes les références à votre projet. Cliquez Oui et constatez que la nouvelle classe a également été renommée en **Personne**. Si ce n'est pas le cas, faites la modification vous-même.

4. Dans le corps de la classe *Personne*, créez 3 régions nommées **Attributs**, **Constructeurs**
et **Méthodes** selon la syntaxe suivante :

```
#region NomDeRégion

#endregion
```

5. Dans la région **Attributs**, créez quatre champs publics : *nom*, *prenom* et *adresse* de type **string**, *age* de type **int**.

6. Dans la région **Constructeurs**, créez trois constructeurs publics pour la classe. Les signatures
de ces constructeurs seront les suivantes :

```
public Personne()

public Personne(string nom, string prenom)

public Personne(string nom, string prenom, string adresse, int age)
```

7. Dans la région **Méthodes**, créez une méthode publique *GetInfo*, retournant une chaîne de caractères et n'acceptant aucun paramètre. La chaîne résultat sera du type « *nom prénom*, *xx* ans, habite *adresse* » si *age* et *adresse* sont connus (respectivement différents de 0 et ""), sinon elle sera du type « *nom prénom*, *aucune autre information disponible* ».

8. Enregistrez votre travail.

- **Utilisation d'une bibliothèque de classes**

1. Ajoutez une référence à votre nouvelle bibliothèque de classe dans le projet du jeu :

Dans *l'Explorateur de Solutions*, effectuez un clic-droit sur le noeud **Dépendances** du projet **JeuPoM** et choisissez *Ajouter une référence de projet*. Cochez la case correspondant à vitre projet **Info** et cliquez **OK**.

2. Déplacez tout le contenu actuel de la méthode *static void Main* dans une nouvelle méthode *static void LeJeu(Personne p)*.
De façon à ce que le compilateur reconnaisse la classe *Personne*, ajoutez l'instruction `using Info;` à la fin des **using** déjà présents. Sans cela, il faudrait alors écrire *Info.Personne* au lieu de *Personne*.

3. Dans le *static void Main* à présent vide, déclarez une variable *joueur* de type *Personne* et initialisez-la à **null**.

4. Utilisez la fonction *GetString* pour demander au joueur s'il souhaite se présenter. S'il accepte, instanciez la classe *Personne* dans la variable *joueur*, en demandant à l'utilisateur son nom
et son prénom.

5. Appelez la méthode *LeJeu* en passant en paramètre la variable *joueur*. Celle-ci contiendra soit la valeur *null* soit un objet de type *Personne* si le joueur a souhaité nous donner son nom et son prénom.

6. Localisez le commentaire **TODO : Exercice 2**.

7. Si le paramètre *p* est différent de **null**, utilisez *nom* et *prenom* de *Personne* pour spécifier le nom du fichier de sauvegarde de l'historique (en ajoutant également la date). Autrement, le joueur sera interrogé comme précédemment sur l'enregistrement ou l'affichage de l'historique.

Nous vous laissons libre sur le nom du fichier qui sera utilisé mais l'idée serait un nom du type : `nom_prenom_jour_mois_annee.txt`

8. Testez le programme.

Module 7 – Encapsulation de données et de méthodes

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- encapsuler des données ;
- mettre en œuvre des membres statiques ;
- créer des méthodes d'extension ;

Exercice 1
Encapsulation de données

Dans cet exercice, vous allez masquer certaines données d'une classe et assurer leur lecture
par le biais d'un accesseur.

- **Privatisation d'un membre de classe**

1. Ouvrir la solution du dossier **.\Ateliers\Mod07\Starter\JeuPoM** dans Visual Studio.

2. Localisez le commentaire **TODO : Exercice 1** dans la classe *Partie*. Modifiez la déclaration du membre *valeur* de façon à le rendre inaccessible en dehors de la classe.

3. Dans la région **Méthodes** de la class *Partie*, écrivez une méthode publique *GetValeur* qui retournera la valeur du membre privée.

4. Enregistrez et vérifiez que votre code ne génère pas d'erreur de compilation.

Exercice 2
Les membres statiques

Dans cet exercice, vous allez mettre en œuvre des membres statiques pour la classe *Partie*, afin de gérer de manière automatique le nombre de parties.

- **Création d'un membre statique**

1. Localisez le commentaire **TODO : Exercice 2**.

2. Créez une région de code nommée **Membres statiques**.

3. Dans la région de code **Membres statiques**, créez un champ privé statique entier *compteurParties* initialisé à 0.

4. A la suite du code précédent, créez une méthode statique publique permettant de récupérer
la valeur de *compteurParties*. Cette méthode aura la signature suivante :

```
public static int GetNbParties()
```

5. Modifiez le constructeur *Partie(int)* de sorte à incrémenter le membre statique privé *compteurParties* à chaque nouvelle instanciation de la classe.

- **Utilisation d'un membre statique**

1. Grâce aux membres créés dans la classe *Partie*, il n'est à présent plus nécessaire de compter manuellement le nombre de parties dans le programme principal. Supprimez donc toute référence
à la variable *nbPartie*, en la remplaçant lorsque cela est nécessaire par le membre statique *Partie.getNbParties()* (attention au décalage N° partie/Indice de tableau). Nous modifierons les méthodes AfficheHistorique dans l'exercice suivant.

2. Testez le programme.

Exercice 3
Méthodes d'extension

Dans cet exercice, vous allez réécrire les méthodes de gestion de l'historique, afin de les transformer en méthode d'extension du type *Partie[]*.

- **Création de méthodes d'extension**

1. Ajouter une nouvelle class au projet **JeuPoM** et nommez-la *Utilitaire*.cs.

2. Localisez le commentaire **TODO : Exercice 3.1** du fichier *Program.cs* et déplacez les deux méthodes AfficheHistorique dans la nouvelle classe *Utilitaire*.

3. Modifiez les deux méthodes *afficheHistorique* de façon à ce qu'elles deviennent des méthodes d'extension du type *Partie[]*. Pour cela :

- Déclarez les méthodes comme publiques et statiques
- Supprimer le paramètre *compteur* (nous allons pouvoir utiliser le membre statique de l'exercice précédent dans le corps des méthodes pour le remplacer)
- Ajouter l'indicateur *this* au paramètre de type *Partie[]*.

4. Effectuez les dernières corrections nécessaires, sans oublier les clauses *using* obligatoires pour le bon fonctionnement du code.

5. Localisez le commentaire **TODO : Exercice 3.2** du fichier *Program.cs* et modifiez les appels à A*fficheHistorique*, afin de tirer parti des nouvelles méthodes d'extension (invoquées sur la variable *historique*).

6. Testez le programme.

- Module 8 – Héritage de classes et implémentation d'interfaces

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- mettre en œuvre l'héritage de classe ;
- implémenter une interface ;

Exercice 1
Syntaxe d'héritage et d'interface en C#

Dans cet exercice, vous allez devoir étudier des syntaxes C# afin de comprendre leur fonctionnement ou bien pour corriger leurs erreurs.

- **Trouver les erreurs de syntaxe dans le code suivant :**

```
interface IToken
{
        string Name( );
        int LineNumber( ) { return 42; }
        string name;
}
class Token
{
        string IToken.Name( ) { .... }
        static void Main( )
        {
                IToken t = new IToken( );
        }
}
```

- **Quel est le résultat d'exécution du code suivant :**

```
class A {
        public virtual void M() { Console.Write("A"); }
}
class B: A {
        public override void M() { Console.Write("B"); }
}
class C: B {
        new public virtual void M() { Console.Write("C"); }
}
class D: C {
        public override void M() { Console.Write("D"); }
        static void Main( ){
                D d = new D(); C c = d; B b = c; A a = b;
                d.M(); c.M(); b.M(); a.M();
        }
}
```

Exercice 2
Héritage de classe

Dans cet exercice, vous allez mettre en œuvre l'héritage à partir de la classe *Personne* dans deux cadres d'application. Dans un premier temps, vous créerez une nouvelle classe *Joueur* héritant
de *Personne*, de façon à terminer le Jeu du Plus ou Moins. Dans un deuxième temps, vous créerez une nouvelle classe *Employe* héritant de *Personne*, qui servira de base au deuxième programme mis en œuvre pendant cette formation.

- **Création de la classe Joueur**

1. Ouvrez la solution du dossier **.\Ateliers\Mod08\Starter\JeuPoM** dans Visual Studio.

2. Ajoutez une nouvelle classe **Joueur.cs** dans le projet **JeuPoM**.

3. Indiquez que la classe *Joueur* hérite de la classe *Personne*. Il vous sera nécessaire de résoudre
le nom par l'ajout de la clause **using** adéquate.

4. Ajoutez un membre publique à la nouvelle classe, nommé *parties* de type *Partie[]*.

5. Définissez un constructeur unique pour la classe *Joueur*, prenant en paramètre le nom et le prénom du joueur, et qui réalisera également l'initialisation du tableau de parties avec un nombre d'éléments à 20.

6. Localisez le commentaire **TODO : Exercice 2.1** dans le fichier *Program.cs*.

7. Modifiez le code de façon à utiliser la classe Joueur à la place de la classe *Personne*.

9. Supprimez la question à l'utilisateur, il devra être systématiquement être reconnu à partir de maintenant.

10. Localisez le commentaire **TODO : Exercice 2.2**.

11. Modifiez la déclaration de la méthode *LeJeu* pour prendre en compte le fait que la variable est
à présent de type *Joueur*. Il est à noter que nous aurions pu laisser le type *Personne* (un joueur est une personne).

12. Dans le corps de la méthode *LeJeu*, supprimez toute référence au tableau local *historique*,
pour utiliser à la place le membre *parties* de la classe Joueur.

13. Le joueur étant obligatoirement connu, il n'est également plus nécessaire de tester la validité
de la variable *p* (commentaire **TODO : Exercice 2.3**). Supprimez tout le code inutile et conservez uniquement la sauvegarde dans un fichier nommé automatiquement en fonction du joueur.

14. Testez le programme.

- **Création de la classe Employé**

1. Ajoutez une nouvelle classe **Employe.cs** au projet **Info**.

2. Indiquez que la classe *Employe* hérite de la classe *Personne*, et indiquez sa portée comme **public**.

3. Dans le corps de la classe *Employe*, créez 3 régions nommées **Attributs**, **Constructeurs** et **Méthodes** selon la syntaxe suivante :

```
#region NomDeRégion

#endregion
```

4. Dans la région **Attributs**, créez les champs suivants: *numero* (privé, entier), *fonction* (public, chaîne de caractères) et *salaire* (privé, double).

5. Dans la région **Constructeurs**, créez deux constructeurs publics pour la classe. Les signatures
de ces constructeurs seront les suivantes :

```
public Employe(string nom, string prenom, string adresse, int age, string fonction,
        double salaire)

public Employe(Personne p, string fonction, double salaire)
```

6. Dans la région **Méthodes**, nous pourrions écrire une version polymorphe de la méthode **GetInfo** de la classe *Personne*. Il faudrait pour cela ajouter la déclaration **virtual** à la méthode de la classe mère et créer une nouvelle méthode dans la classe enfant qui serait **override**. Toutefois, le comportement de ces méthodes est identique à ce que propose la méthode polymorphe **ToString** de la classe *Object*. Nous allons donc écrire dans nos deux classes la version polymorphe nécessaire à notre problématique.

Dans la classe *Personne*, dupliquez la méthode *GetInfo* et renommez la copie *ToString*. Ajoutez le mot-clé **override** à la déclaration.

Dans la classe *Employe*, ajoutez la méthode *ToString* en réutilisant la même signature que précédement. Dans le corps de la méthode, retournez une chaîne de caractère, en ajoutant au résultat de *ToString* de la classe mère, la fonction de l'employé.

7. Dans la région **Méthodes**, créez ensuite deux méthodes publiques *Augmentation* et *Affectation*
qui permettront de modifier respectivement *salaire* et *fonction*. Le paramètre de la méthode *Augmentation* sera le montant de l'augmentation, celui de la méthode *Affectation* sera la nouvelle fonction.

8. Enregistrez votre travail et compilez votre projet (menu **Générer** puis **Regénérer la solution**). Corrigez toutes les erreurs si nécessaire.

- **Test de la classe Employé**

1. Ajoutez un nouveau projet nommé **GestionRH**, de type **Application Console**, dans la solution **JeuPoMSIn**. Faites attention à bien choisir une application de type .NET (et non .NET Framework) en C#.

2. Ajoutez la référence au projet *Info* dans ce nouveau projet (clic-droit sur **Dépendances**).

3. Remplacez le code du fichier **Program.cs** par le contenu du fichier **Extrait.txt** du dossier **.\Ateliers\Mod08\Starter**.

4. Dans le corps du *static void Main*, déclarez et instanciez un nouvel employé, et vérifiez le bon fonctionnement des méthodes publiques de la classe.

5. Définissez le projet **GestionRH** comme projet par défaut dans Visual Studio : clic-droit sur le projet puis **Définir en tant que projet de démarrage**.

6. Lancez le débogage et utilisez les techniques de débogage comme le pas à pas détaillé et les indicateurs info-bulles afin de valider que votre classe **Employe** fonctionne bien.

- Module 9 – Gestion de la durée de vie des objets et contrôle des ressources

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- mettre en œuvre interface IDisposable ;
- utiliser une classe implémentant l'interface IDisposable ;

Exercice 1
Implémentation de l'interface IDisposable

Dans cet exercice, vous allez mettre en œuvre l'implémentation de l'interface **IDisposable** dans
la classe *Employe* créée dans le module précédent.

- **Implémentation de l'interface IDisposable**

1. Ouvrez la solution du dossier .**\Ateliers\Mod09\Starter\GestionRHSln** dans Visual Studio Code et acceptez l'ouverture de l'espace de travail.

2. Ouvrez le fichier de code de ma classe **Employe**..

3. Indiquez que la classe *Employe* implémente l'interface *IDisposable*.

4. Grâce aux actions rapides de **Visual Studio** (la petite ampoule en face de la ligne), implémentez l'interface *IDisposable* avec le modèle Dispose. Ce mode d'implémentation nous permet de récupérer tous les élements nécessaires à notre code, contrairement à une implémentation sans modèle qui ne générerait que le membre *Dispose* de l'interface.

5. Notez qu'un champ *disposedValue* a été ajouté dans la région **Attributs** et deux méthodes *Dipose(bool disposing)* et *Dispose()* dans la région **Méthodes**. Vous pourrez également constater la présence en commentaire du destructeur *~Employe*.

6. Décommentez le destructeur de la classe *Employe*.

7. Dans la méthode *Dispose(bool disposing)*, repérez le commentaire **TODO : supprimer l'état managé**. Pour rappel, si le paramètre *disposing* est vrai, il s'agit d'une desctruction manuelle de l'objet (lorsqu'on le souhaite), alors qu'une valeur *faux* indiquerait une destruction gérée par le **Garbage Collector**.
Affectez la valeur **null** à la fonction de l'employé.

8. Ouvrez le fichier *Program.cs* du dossier **GestionRH**.

9. Positionnez un point d'arrêt sur la dernière ligne de code de la méthode *static void Main* (*Console.WriteLine(e);*).

10. A la suite de la ligne pointée par le point d'arrêt, utilisez une structure **using** pour instancier
un nouvel *Employe*, et dans laquelle vous mettrez en œuvre l'affichage sur la console des informations de l'employé.

11. Lancez le débogage du programme (menu **Exécuter**, puis **Démarrer le débogage**).

12. Ajoutez un espion pour la variable de type *Employe* déclarée dans le bloc **using**. Constatez le message du panneau Espion, indiquant que la variable n'existe pas dans le contexte actuel.

19. Réalisez un **pas à pas principal** (F10) pour avancer sur la première ligne du bloc **using**. Constatez qu'à présent la variable contient **null**. Avancez en **pas à pas principal** pour atteindre l'accolade fermante du bloc **using**.

20. Réalisez un **pas à pas détaillé** (F11). Notez que le programme appelle la méthode *Dispose()*
de la classe *Employe*. Continuez le **pas à pas détaillé** pour constater l'appel de la méthode *Dispose(bool)* et la suppression de la *fonction* de *l'Employe*.

21. Poursuivez le **pas à pas détaillé** jusqu'à ce que le code du *static void Main* s'affiche à nouveau dans **Visual Studio**. L'indicateur d'exécution devrait pointer l'accolade fermante du *static void Main*. Constatez dans le panneau Espion que la variable n'existe plus.

22. Terminez l'exécution en appuyant sur F5.

- **Création de classe implémentant IDisposable**

1. Ajoutez une nouvelle classe **Entreprise.cs** au projet **Info**.

2. Modifiez la déclaration de la classe **Entreprise** pour la rendre **public**.

3. Dans le corps de la classe *Entreprise*, créez 3 régions nommées **Attributs**, **Constructeurs**
et **Méthodes** selon la syntaxe suivante :

```
#region NomDeRégion

#endregion
```

4. Dans la région **Attributs**, créez les champs suivants : *effectif* (public, entier), *nom* (public, chaîne
de caractères) et *employes* (public, tableau d'*Employe*).

5. Dans la région **Constructeurs**, créez deux constructeurs publics pour la classe. Les signatures
de ces constructeurs seront les suivantes :

```
public Entreprise(string nom)

public Entreprise()
```

6. Dans la région **Méthodes**, créez une méthode publique E*mbauche*, sans valeur de retour et acceptant un paramètre de type *Employe*. Dans le corps de la méthode, vérifiez que le nombre maximal d'employés n'est pas atteint, et dans ce cas, ajoutez l'employé passé en paramètre au tableau *employes*, en vous servant du champ *effectif* comme compteur pour le tableau. Pour le cas où le nombre maximal d'employés est atteint, déclenchez une **Exception**.

7. Dans le fichier *Employe.cs*, localisez la région **Méthodes** et créez une nouvelle méthode ayant pour signature : *internal void setNumero(int matricule)*. Utilisez cette méthode pour affecter le paramètre *matricule* au champ privé *numero*.

8. Dans la méthode *embauche* de la classe *Entreprise*, appelez la méthode *setNumero* sur l'employé nouvellement ajouté au tableau *employes*, pour lui affectez son numéro matricule

(correspondant
à l'indice de position de l'objet dans le tableau plus un).

Ex : pour le premier employé (position 0 dans le tableau), son numéro matricule est 1
pour le deuxième employé (position 1 dans le tableau), son numéro matricule est 2

=> on constate donc que le numéro matricule est le nombre d'éléments dans le tableau juste après le recrutement de l'employé (*this.effectif*)

9. Recopiez lés éléments du modèle **Dispose** de la classe *Employe* vers la classe *Entreprise* (attribut *disposeValue*, méthodes *Dispose()*, *Dispose(bool disposing)* et *~Employe*). Renommez le destructeur *~Employe* en *~Entreprise*.

10. Localisez la méthode *Dispose(bool)* de la classe *Entreprise*, et remplacez la ligne *fonction = null;* par le code nécessaire à l'appel de la méthode *Dispose()* sur tous les objets *Employe* du tableau *employes*.

11. Localisez la méthode *static void Main* dans le fichier *Program.cs*.

12. A la suite du bloc **using** mis en œuvre précédemment, déclarez un nouveau bloc **using** afin d'instancier la classe *Entreprise*.

13. Dans le nouveau bloc **using**, utilisez la méthode E*mbauche* de la classe *Entreprise* pour ajouter deux employés : le premier sera l'employé *e* instancié au début de la méthode, le deuxième sera créé directement dans l'appel de la méthode *Embauche*.

14. Utilisez une structure de boucle pour visualiser les informations de chaque employé (par un *Console.WriteLine()*) grâce au tableau *employes* de l'instance d'*Entreprise*.

15. Placez un point d'arrêt sur l'accolade fermante du bloc **using** et lancez le débogage du programme (menu **Exécuter**, puis **Démarrer le débogage**). Surveillez la variable *e*. Que constatez-vous en sortie du bloc **using** ?

La variable *e* contient toujours la référence de la variable *Employe*, toutefois, celle-ci a été détruite par l'appel de la méthode *Dispose* (*fonction* a été définie à **null**). Cela signifie donc que tout employé rattaché à l'entreprise ne pourra plus être manipulé après la destruction de celle-ci.

Module 10 – Encapsulation avancée

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- mettre en œuvre des propriétés pour une classe C# ;
- créer un indexeur ;
- comprendre la syntaxe de surcharge d'opérateur ;

Exercice 1
Création de propriétés

Dans cet exercice, vous allez complétez l'application de Gestion des Ressources Humaines, en mettant en œuvre des propriétés de classe soit pour rendre accessibles certaines informations privées, soit pour calculer certaines valeurs nécessaires au fonctionnement du programme.

- **Mise en œuvre de la liste des employés**

1. Ouvrez la solution du dossier **.\Ateliers\Mod10\Ex1\Starter\GestionRHSln** dans Visual Studio.

2. Localisez le commentaire **TODO : Exercice 1.1** dans le fichier *Program.cs*.

3. Mettez en œuvre une structure d'itération pour afficher les informations de tous les employés de l'entreprise sur la *Console*.

4. Testez le programme.

- **Mise en œuvre du recrutement d'un nouvel employé**

1. Localisez le commentaire **TODO : Exercice 1.2**.

2. Utilisez les fonctions G*etString* et *GetEntier* pour demander à l'utilisateur les informations nécessaires à la création d'un nouvel *Employe* (*nom*, *prenom*, *adresse*, *age*, *fonction* et *salaire*).
Par facilité, le salaire sera considéré comme un **entier**.

3. Instanciez un nouvel *Employe* à l'aide des valeurs précédemment récupérées et utilisez la méthode E*mbauche* de la variable *ms* pour ajouter le nouvel *Employe*.

4. Testez le programme.

Remarque : la liste des employés chargée à chaque exécution n'est pas mise à jour suite à ce recrutement. Cela pourrait constituer un exercice supplémentaire que nous n'avons pas envisagé ici.

- **Récupération d'un employé existant**

1. Localisez le commentaire **TODO : Exercice 1.3**.

2. Utilisez la fonction G*etEntier* pour demander à l'utilisateur le numéro de l'employé à modifier.

3. Utilisez cet entier et le tableau *employes* de la variable *ms* pour affecter à la variable *emp* l'objet de type *Employe* correspondant. Pour le cas où le numéro spécifié par l'utilisateur ne corresponde pas à un employé existant, redirigez-le vers le **Menu Principal**.

Rappel : le numéro matricule de l'employé est en fait l'indice de position dans le tableau incrémenté de 1.

4. Testez le programme.

- **Création de propriétés**

1. Ouvrez le fichier *Employe.cs* et ajoutez une région de code **Propriétés**.

2. Créez dans cette région, une propriété en lecture seule nommée *Salaire* retournant le *salaire*
de l'employé. Cette propriété sera déclarée comme **internal**.

3. Ouvrez le fichier *Entreprise.cs* et ajoutez une région de code **Propriétés**.

4. Créez dans cette région, une propriété en lecture seule nommée *ChargeSalariale*, retournant
la somme des salaires de tous les employés de l'entreprise. Cette propriété sera déclarée comme **public**.

5. Localisez le commentaire **TODO : Exercice 1.4** dans le fichier *Program.cs*.

6. Affichez les valeurs suivantes : nombre d'employés, charge salariale et salaire moyen (converti
en entier).

7. Testez le programme.

Remarque : les exemples de données fournis donnent les valeurs ci-dessous.

Nombre d'employés	Charge salariale	Salaire moyen
8	32901	4112

Exercice 2
Création d'une propriété Indexeur

Dans cet exercice, vous allez ajouter un indexeur à la classe *Entreprise* de façon à pouvoir utiliser
le formalisme tableau sur une instance de cette classe.

- **Mise en œuvre d'une propriété indexeur**

1. Ouvrez la solution du dossier **.\Ateliers\Mod10\Ex2\Starter\GestionRHSIn** dans Visual Studio.

2. Dans le code de la classe Entreprise, changez le modificateur d'accès du membre *employes* en **private**.

3. Ajoutez dans la région de code **Propriétés**, une nouvelle propriété indexeur permettant l'accès

en lecture et écriture au tableau *employes*.
Dans la section **get** de l'indexeur, vérifiez l'indice fourni en paramètre de sorte qu'il permette effectivement d'accéder à un objet **non null** du tableau *employes*, et retournez la valeur **null** sinon.

Dans la section **set** de l'indexeur, une valeur d'indice entre 0 et *effectif-1* modifiera directement l'objet correspondant du tableau *employes*. Autrement déclenchez une exception de type **IndexOutOfRangeException**.

4. Corrigez le code du fichier *Program.cs* de sorte à remplacer les appels à *ms.employes[]* par *ms[]*.

5. Compilez la solution. Que constatez-vous ?

Le compilateur renvoie une erreur indiquant que l'instruction *foreach* ne peut s'exécuter sur le type *Entreprise*, car celui-ci n'implémente pas l'interface **IEnumerable**. Nous envisagerons ce problème ultérieurement.

6. Remplacez la structure d'itération **foreach** utilisée dans le *static void Main* du fichier *Program.cs* par une structure **for(;;)**.

7. Testez le programme.

Exercice 3
Surcharge d'opérateur (facultatif)

Dans cet exercice, vous allez devoir corriger des syntaxes de surcharge d'opérateur.

- **Trouvez les erreurs de syntaxe**
    - public bool operator != (Time t1, Time t2) { ... }
    - public static operator float(Time t1) { … }
    - public static Time operator += (Time t1, Time t2) { ... }
    - public static bool Equals (Object obj) { ... }
    - public static int operator implicit(Time t1) { ...}

• Module 11 – Découplage de méthodes et gestion d'événements

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- mettre en œuvre un délégué ;
- créer et gérer un événement ;

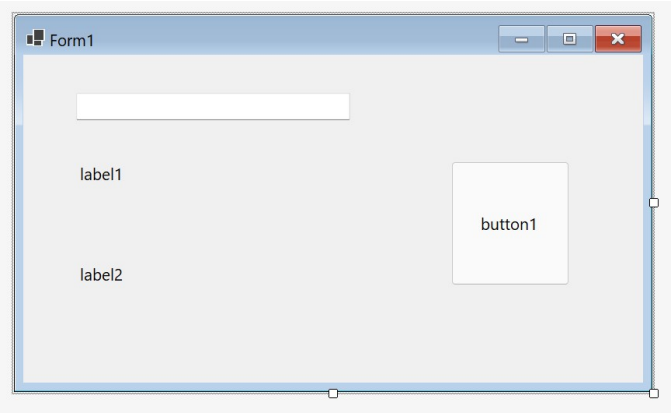
Exercice 1
Application Graphique et événement

A travers cet eercice, nous allons reprendre l'idée du petit jeu du plus ou moins, mais cette fois à l'aide d'une interface graphique. Pour cela, nous mettrons en œuvre une application de type Windows Forms.

- **Création de l'application**

1. Lancez Visual Studio et créez un nouveau projet appelé **JeuPoMGUI** de type **Application Windows Forms**.

2. A l'aide du panneau **Boîte à Outils**, ajoutez à l'interface de l'application une zone de texte (**TextBox**), deux étiquettes (**Label**) et un bouton (**Button**). Nous vous proposons la disposition suivante :



3. A l'aide du panneau **Propriétés**, modifiez la propriété **Name** de ces contrôles pour les appeler comme suit :
TextBox1 => txtSaisie ; Label1 => lblInfo ; Label2 => lblTentative ; Button1 => cmdMain

4. Modifiez le texte du bouton **cmdMain** et affectez-lui la valeur « Lancez une partie ». De même modifiez le texte de l'objet **Form1** avec la valeur « Jeu ». Définissez la propriété **AcceptButton** de l'objet **Form1** à **cmdMain** (permet de cliquer le bouton par la touche Entrée).

5. Double cliquez le bouton pour atteindre la méthode de gestion d'événement située dans le fichier de code associé à l'inteface graphique. Notez que l'affectation au délégué se fait dans un fichier « caché » nommé *Form1.Designer.cs*.

6. Avant d'écrire le code de gestion du clic du bouton, déclarez les membres suivants en tant que membres de la classe **Form1** (à l'exécution, la classe sera instanciée pour créer la fenêtre applicative) : **valeur** et **tentative**, toutes deux de type entier.

7. Dans la méthode **cmdMain_Click**, affectez à la variable **valeur** un nombre aléatoire compris entre 0 et 100 (utilisez pour cela la classe Random comme dans les premiers exercices) et 0 à la variable **tentative**.

8. Modifiez les propriétés Text de lblInfo et de cmdMain comme suite :

```
lblInfo.Text = "Veuillez saisir un entier";
cmdMain.Text = "Ok" ;
```

9. Plutôt que de travailler avec 2 boutons et de devoir jouer sur leur visibilité ou activation, nous allons tout simplement modifier le gestionnaire d'événement du bouton pour transformer son comportement. Pour cela, utilisez l'opération **-=** sur l'événement **Click** de **cmdMain** pour retirer la méthode actuelle et ensuite l'opération **+=** pour ajouter un nouveau gestionnaire d'événémnet que nous nommerons **Jeu_Click**.

10. En utilisant les actions rapides de Visual Studio, faîtes générer la nouvelle méthode **Jeu_Click**. Remplacez le code existant par le code nécessaire pour comparer la saisie utilisateur dans la zone de texte à la valeur secrète. Utilisez lblInfo pour afficher le message correspondant (trop petit ou trop grand) et lblTentative pour afficher le nombre de tentative (en pensant à l'incrémenter à chaque tentative).
NOTES :

a. vous pouvez ou non choisir de gérer le fait que l'utilisateur saisisse autre chose qu'un entier. La correction vous proposera une version du code faisant intervenir un composant graphique pour signaler une erreur de saisie.

b. pour aider le joueur, vous pouvez ajouter les deux commandes suivantes dans le cas où il n'a pas trouvé la bonne valeur :

```
txtSaisie.Focus(); // Renvoi le curseur dans la zone ; possible de s'en servir au lancement du jeu
txtSaisie.SelectAll(); // Sélectionne tout le contenu
```

11. Lorsque l'utilisateur aura trouvé la bonne valeur, affichez une boîte de dialogue pour lui signaler qu'il a gagné :

```
MessageBox.Show("Message","Jeu",MessageBoxButtons.OK, MessageBoxIcon.Information)
```

12. A la suite du code précédent, faites en sorte de réintervertir les deux procédures de gestion d'événement et de rétablir les valeurs de propriété **Text** adéquates (état initial de l'application).

13. Testez votre programme.

Remarque : Afin de simplifier les syntaxes, vous allez utiliser des **délégués génériques** fournit
par .Net (présentés dans le chapitre suivant), ce qui évitera la nécessité de déclarer des **délégués** spécifiques à l'application.

Exercice 2
Mise en œuvre d'un délégué

Dans cet exercice, vous allez créer une méthode acceptant un **délégué** en paramètre afin de laisser au code appelant le choix de la méthode à mettre en œuvre. Lors d'une nouvelle affectation de poste, nous allons ainsi laisser la possibilité à l'utilisateur de la classe *Employe* de fournir un code exécutable (une expression Lambda par exemple) permettant de traiter un message sur le changement d'affectation de l'employé (ici affichage sur la *Console*).

- **Mise en œuvre d'un délégué**

1. Ouvrez la solution du dossier **.\Ateliers\Mod11\Starter\GestionRHSln** dans Visual Studio.

2. Localisez le commentaire **TODO : Exercice 1.1** dans le fichier *Employe.cs*.

3. Créez une surcharge à la méthode A*ffectation*, prenant un deuxième paramètre de type **Action<String>** nommé *methode*.

Remarque : l'utilisation du **delegate Action<String>**, revient à déclarer un **délégué** :

```
public delegate void action(String msg) ;
```

4. Avant de modifier le membre *fonction*, appelez le **délégué** *methode* en lui passant en paramètre une chaîne de caractères du type « *nom prenom*, *fonction* devient *nlleAffectation* ».

5. Localisez le commentaire **TODO : Exercice 1.2** dans le fichier *Porgram.cs*.

6. Modifiez l'appel de la méthode *affectation*, pour utiliser la surcharge que vous venez de créer
en passant en deuxième paramètre une **expression Lambda** pour afficher le message en majuscule sur la Console.

7. Testez le programme.

Exercice 3
Création et gestion d'un événement

Dans cet exercice, vous allez créer un événement qui sera déclenché lorsque le nombre d'employé de l'entreprise sera modifié, afin d'indiquer le nombre de postes restants. De plus, vous gérerez
cet événement dans l'application de Gestion des Ressources Humaines pour afficher le message
à l'utilisateur.

- **Création d'un événement**

1. Ajoutez à la classe *Entreprise* un événement *InfoEffectif*, respectant le formalisme des événements .NET. Pour cela, vous mettrez en œuvre le **délégué Action** avec deux paramètres : le premier de type **Object**, le second de type **EntEventArgs** (que vous créerez ensuite).

Votre syntaxe devrait ressembler au code suivant :

```
public event Action<Object,EntEventArgs> InfoEffectif=null ;
```

2. Pour créer la classe *EntEventArgs*, dans la syntaxe précédente, cliquez le terme *EntEventArgs* pour qu'apparaisse la petite ampouel d'action rapide en début de ligne. Cliquez dessus et choisissez **Générez le type 'EntEventArgs' -> Générer class 'EnteEventArgs' dans un nouveau fichier**.

3. Dans le fichier *EntEventArgs.cs* que **Visual Studio Code** vient de générer, indiquez que la classe *EntEventArgs* hérite de la classe **EventArgs**, sans oublier d'ajouter la directive *using System;* .

4. Ajoutez à cette classe, un membre privé de type **int** nommé *nombrePosteRestant*.

5. Ajoutez une propriété publique en lecture seule **Information** de type **string**, qui retournera une chaîne de caractères de type « Il reste *n* poste(s) de libres », où *n* sera le nombre de poste restant.

5. Ajoutez un constructeur à la classe, prenant un entier *nbPosteRestant* en paramètre, qui appellera le constructeur de la classe de base, et initialisera le champ *nombrePosteRestant* avec la valeur du paramètre.

6. Avant d'utiliser notre nouvelle classe, nous allons effectuer une petite modification sur la classe *Entreprise*, de sorte à founir le nombre maximum d'employés dès l'instanciation.

Ajoutzr à la classe *Entreprise*, un membre privé de type **int** nommé *maxEmploye* et en ajoutant un nouveau paramètre au constructeur de la classe, intialisez ce nouveau membre. Utilisez ensuite ce membre dans la création du tableau d'employés.

N'oubliez pas de modifier également le constructeur vide ainsi que le programme principal.

6. Localisez le commentaire **TODO : Exercice 2.1** dans le fichier *Entreprise.cs*.

7. Testez si le membre *InfoEffectif* est **non null**, et dans ce cas, déclenchez l'événement en indiquant en paramètre l'objet en court et une nouvelle instance de la classe **EntEventArgs** (le nombre de poste restant dans l'entreprise étant passé en paramètre du constructeur par différence entre *maxEmploye* et *effectif*).

8. Enregistrez votre travail.

- **Gestion de l'événement**

1. Localisez le commentaire **TODO : Exercice 2.2** dans le fichier *Program.cs*.

2. Ajoutez un gestionnaire d'événement à la variable *ms*, permettant d'afficher sur la console la propriété *Information* de l'objet *EntEventArgs*.
Pour cela, utilisez une nouvelle instance de la classe *delegate*
Action<Object,EntEventArgs>
en lui passant en paramètre une **expression Lambda** permettant de réaliser l'affichage demandé.

Votre syntaxe devrait ressembler au code suivant :

```
ms.InfoEffectif +=
        new Action<Object,EntEventArgs>((o,e) => Console.WriteLine(e.Information)) ;
```

Mais peut également être abrégé en :

```
ms.InfoEffectif +=  (o,e) => Console.WriteLine(e.Information) ;
```

Remarque : Le gestionnaire d'événement est ajouté après le chargement des données du fichier texte afin de ne pas afficher le message à chaque employé initialement présent dans l'entreprise.

3. Testez le programme.

- ## Module 12 – Utilisation des collections et construction de types génériques

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- utiliser une collection générique ;

Exercice 1
Utilisation d'une collection

Dans cet exercice, vous allez utiliser une collection du .Net Framework afin de remplacer l'actuel tableau permettant de stocker les employés de l'entreprise. Les modifications apportées au code
de la classe Entreprise seront transparentes pour l'application finale.

- **Utilisation d'une collection**

1. Ouvrez la solution du dossier **.\Ateliers\Mod12\Starter\GestionRH** dans Visual Studio.

2. Localisez le commentaire **TODO : Exercice 1.1** dans le fichier *Entreprise.cs*.

3. Modifiez la déclaration du membre *employes* pour utiliser le type **List<Employe>** à la place d'un tableau d'*Employe*. La classe **List<Employe>** appartenant à l'espace de nom **System.Collections.Generic**, ajoutez l'instruction **using** correspondante en début de code.

4. Supprimez la déclaration du membre *effectif*. Il n'est plus nécessaire car la classe **List<Employe>** nous fournit le nombre d'élément par sa propriété *Count*.

5. Dans la région de code **Propriétés**, ajoutez une nouvelle propriété *Effectif* en lecture seule de type entier, retournant le nombre d'élément du membre *employes*. Remplacez toute référence à *effectif* par la nouvelle propriété *Effectif* à la fois dans la classe *Entreprise* et dans le programme principal.

6. Localisez le commentaire **TODO : Exercice 1.2**.

7. Modifiez l'instanciation du membre *employes* en instanciation de la classe **List<Employe>**. De ce fait, nous pourrions considérer qu'il n'est plus nécessaire de conserver la taille maximale de l'entreprise (champ *maxEmploye*). Toutefois, nous allons le conserver de sorte à limiter volontairement la taille de l'entreprise (ce ne sera donc plus une contrainte technique mais une contrainte métier).

8. Supprimez l'initialisation du membre *effectif*.

9. Localisez le commentaire **TODO : Exercice 1.3**.

10. Modifiez la méthode *embauche* en tenant compte des fonctionnalités de la classe **List<Employe>** (méthode **Add** pour ajouter un nouvel élément) et en imposant une limite d'embauche en fonction de notre champ *maxEmploye*.

11. Testez le programme.

Remarque : il est important de noter que par le principe d'encapsulation que nous avons appliqué jusque là dans cette classe, aucune autre modification n'est nécessaire et que le programme principal fonctionne toujours suite au remplacement du tableau par la collection. Cela n'a pas été le cas avec le champ *effectif* devenu une propriété *Effectif* qui nous a obligé à modifier le programme principal. Nous aurions pu créer la propriété en reprenant le nom du champ (à savoir sans la majuscule) mais cela n'aurait pas respecté les conventions de nommage .NET.

De plus, si nous avions utilisé une classe non générique comme la classe **ArrayList**, il nous aurait fallu corriger la propriété *ChargeSalariale* et l'indexeur. En effet, lors de l'extraction d'un objet depuis une instance d'**ArrayList**, nous n'aurions pas obtenu une instance d'*Employe* mais un simple *object* qu'il aurait fallu convertir en tant qu'*Employe*.

- ## Module 13 – Construire et énumérer une classe de collection personnalisée

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- comprendre le concept de programmation asynchrone
- implémenter l'interface IEnumerable ;
- appliquer une syntace de niveau supérieur ;

Exercice 1
Programmation Asynchrone

Dans cet exercice vous allez mettre en évidence le problème de la programmation synchrone et découvrir comment implémenter une approche asynchrone.

1. Ouvrez la solution du dossier **.\Ateliers\Mod13\Starter\AsyncProg** dans Visual Studio.

L'application est de type WPF et n'est constituée que d'un simple bouton et d'une étiquette. Lors du clic sur le bouton, un traitement long est simulé à l'aide de la méthode **Thread.Sleep** qui mets en pause le thread d'exécution.

2. Lancez le débogage de l'application et cliquez sur le bouton. Vous constaterez alors que l'application n'est pas réactive (impossible de déplacer la fenêtre par exemple) tant que le traitement n'est pas terminé.

3. Arrêtez le débogage une fois que le traitement long est fini.

4. Accédez au code et notez la présence de la méthode **DoWork**.

5. Sélectionnez la ligne contenu dans la méthode **DoWork** et commencez à taper **await**. Tapez un espace et notez que Visual Studio a ajouté le mot clé **async** à la déclaration de la méthode. Complétez la ligne en appelant la méthode **Delay** de la classe **Task**. Comme

précédemment, nous fournirons une valeur de paramètre à 15000 pour simuler un traitement long de 15 secondes.

6. Déplacez la ligne affectant la chaîne "Traitement terminé !" après la ligne que vous venez d'écrire dans la méthode **DoWork**.

7. Testez le programme et constatez que la fenêtre est réactive tout le temps du traitement.

Exercice 2
Implémentation de l'interface IEnumerable

Dans cet exercice, vous allez implémenter l'interface IEnumerable de façon à autoriser la syntaxe **foreach** pour parcourir les éléments de la collection personnalisée *Entreprise* que vous avez construite au fil des ateliers précédents.

- **Implémentation de l'interface IEnumerable**

1. Ouvrez la solution du dossier **.\Ateliers\Mod13\Starter\GestionRHSln** dans Visual Studio.

2. Ouvrez le fichier de code *Entreprise.cs*.

3. Modifiez la déclaration de la classe *Entreprise* de sorte qu'elle implémente l'interface **IEnumerable<Employe>**.

4. En cliquant *IEnumerable* avec votre souris, notez l'apparition de la petite ampoule en début de ligne. Cliquez dessus et choisissez l'option **Implémenter l'interface via 'employes'**. Notez les deux méthodes ajoutées par **Visual Studio Code** : *GetEnumerator* et *IEnumerable.GetEnumerator*.

5. Dans ces deux méthodes, constatez qu'est retourné à chaque fois l'énumérateur de notre instance de collection *employes*. Vous pouvez d'ailleurs simplifier le code en éliminant les conversions et en invoquant directement la méthode *GetEnumerator* de la variable *employes*.

6. Localisez le commentaire **TODO : Exercice 1** dans le fichier *Program.cs*.

7. Remplacez la boucle **for(;;)** par une boucle **foreach** afin de tester l'implémentation de l'interface **IEnumerable**.

8. Testez le programme.

Exercice 3
Utilisation de la syntaxe de niveau supérieur

1. Eliminez du fichier *Program.cs* toutes les instuctions inutiles (*namespace*, *class*, *static void Main*) pour ne plus avoir que les *using*, le code du *static void Main* et à la suite les fonctions utilitaires. *Shift+Tab* vous permettra de réalignez le code proprement.

2. Testez le programme.

Module 14 – Utilisation de LINQ pour interroger des données

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- utiliser LINQ pour interroger des données ;

Exercice 1
Utilisation de requêtes LINQ

Dans cet exercice, vous allez écrire plusieurs requêtes LINQ pour assurer la recherche d'informations personnalisées dans un ensemble de données.

- **Recherche d'employé par nom**

1. Ouvrez la solution du dossier **.\Ateliers\Mod14\Starter\GestionRHSln** dans Visual Studio.

2. Ajoutez la directive *using System.Linq;* au fichier *Entreprise.cs*.

3. Localisez la région de code **Méthodes** de la classe *Entreprise*. Ajoutez une nouvelle méthode dont la signature est la suivante :

```
public List<Employe> RechercheParNom(string nom)
```

4. Dans cette méthode, définissez une requête **LINQ** retournant tous les employés dont le *nom* est celui passé en paramètre en utilisant la syntaxe des opérateurs de requête.

5. A l'aide d'une instruction **return**, renvoyez le résultat de cette requête sous forme de liste à l'aide de la méthode **ToList**.

6. Localisez le commentaire **TODO : Exercice 1.1** dans le fichier *Program.cs*.

7. En utilisant la fonction *GetString*, demandez à l'utilisateur le *nom* qu'il recherche.

8. Appelez la méthode *RechercheParNom* sur l'objet *ms* pour afficher les informations des employés correspondants.

9. Testez le programme.

- **Recherche d'employé par fonction**

1. Localisez la région de code **Méthodes** de la classe *Entreprise*.

2. Ajoutez une nouvelle méthode dont la signature est la suivante :

```
public List<Employe> RechercheParFonction(string fonction)
```

3. Dans cette méthode, en utilisant les méthodes d'extension de **LINQ** *Where* et *ToList*, retournez la liste des employés dont la fonction est celle recherchée (en une seule instruction).

5. Localisez le commentaire **TODO : Exercice 1.2** dans le fichier *Program.cs*.

6. En utilisant la fonction G*etString*, demandez à l'utilisateur la *fonction* qu'il recherche.

7. Appelez la méthode R*echercheParFonction* sur l'objet *ms* pour afficher les informations des employés correspondants.

8. Testez le programme.

- **Recherche de la charge salariale pour une fonction recherchée**

1. Localisez la région de code **Méthodes** de la classe *Entreprise*.

2. Ajoutez une nouvelle méthode dont la signature est la suivante :

```
public (int nombre,double charge) ChargeSalarialeParFonction(string fonction)
```

3. Dans cette méthode, définissez une requête **LINQ** retournant tous les employés dont la *fonction* est celle passée en paramètre ou bien réutilisez la méthode *RechercheParFonction*.

4. Parcourez les employés retournés par la requête précédente pour compter le nombre d'employé et calculer la charge salariale correspondante.

Remarque : il est également possible de connaître le nombre d'employé correspondant à l'aide
de la méthode *Count* de la requête **LINQ** et faire la somme des salaires à l'aide de la méthode *Sum*.

5. Utilisez la clause **return** pour renvoyer un tuple qui sera constitué du nombre d'employés de la fonction et de la somme des salaires de ces employés.

6. Localisez le commentaire **TODO : Exercice 1.3** dans le fichier *Program.cs*.

7. Appelez la méthode *ChargeSalarialeParFonction* sur l'objet *ms*. S'il n'existe pas d'employé
pour cette fonction, indiquez-le à l'utilisateur. Autrement, affichez le nombre d'employé, la charge salariale et le salaire moyen pour la fonction choisie.

8. Testez le programme.

- Module 15 – Développement dirigé par les Tests

Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer un projet de test unitaire dans Visual Studio Code;
- tester une classe métier ;

Exercice 1
Création d'un projet de tests

Dans cet exercice, vous allez créer une classe et mettre en œuvre un projet de test pour valider
le code écrit.

1. Créez un nouveau projet de type Bibliothèque de classe nommé **BoiteAOutils**
2. Renommez la classe créée par défaut en tant que **Calculatrice** et marquez-la comme classe statique.
3. Ajoutez une méthode publique statique à cette classe, nommée **Addition**, retournant un entier. Modifiez les paramètres de la méthode de sorte à pouvoir lui passer un nombre indéterminé d'entiers, qui seront ensuite additionnés dans la méthode. Cette somme sera bien évidement le résultat de la fonction.
4. Par un clic droit sur le nom de la méthode **Addition**, choisissez **Créer des test unitaires**. Validez les options choisies par défaut et laisser Visual Studio générer le code nécessaire.
5. Dans la méthode **AdditionTest**, supprimez le code existant et déclarez deux variables de type **int** : **actual** et **expected**. Affectez 10 à la variable **expected**.
6. Affectez à la variable **actual**, le résultat de la méthode **Addition** de votre classe **Calculatrice**, en passant en paramètre les entiers de 1 à 4.
7. A l'aide de la classe **Assert**, appelez la méthode **AreEqual**, de sorte à comparer les variables **actual** et **expected**.
8. Compilez la solution (Menu **Générer**, **Regénérer la solution**).
9. Lancez les tests (Menu **Test**, **Exécuter tous les tests**) et observez le résultat. Corrigez si besoin le code la méthode **Addition** de sorte à ce que le test réussisse.

Exercice 2
Approche TDD

Dans cet exercice, vous allez mettre en évidence l'approche TDD, en écrivant d'abord le test,
puis ensutie la méthode.

1. Dupliquez la méthode **AdditionTest** de la classe **CalculatriceTest** (en prenant l'attribut [TestMethod()]) et renommez cette copie en **MultiplicationTest**.
2. Modifiez cette nouvelle méthode de sorte à ce que la variable **expected** contienne la valeur 24, mais également pour que la méthode appelée sur la classe **Calculatrice** soit **Multiplication** et non **Addition**.

3. Par les options d'*Actions Rapides et Refactorisation* (la petite ampoule en début de ligne, en ayant au préalable cliqué sur l'élément de syntaxe souligné en rouge, ici *Multiplication*), demandez à Visual Studio de générer la méthode **Calculatrice.Multiplication**.
4. Enregistrez l'ensemble des fichiers modifiés et demandez la réexécution des tests. Constatez l'erreur sur la nouvelle méthode de test.
5. Localisez la nouvelle méthode **Multiplication** de la classe **Calculatrice**, constatez la présence de l'exception **NotImplementedException**.
6. Modifiez le code de sorte à mettre en œuvre un paramètre unique similaire à celui de la méthode **Addition** et remplacez le déclenchement d'exception par le code calculant
   le produit des éléments du tableau passé en paramètre.
7. Réexécutez les tests et assurez-vous de leur réussite.

MS860
Correction des Ateliers

Module 2

Exercice 1

- **Trouver les erreurs de syntaxe dans les propositions suivantes :**

  - if number % 2 == 0 …

Il manque les parenthèses pour encadrer la condition : if (number % 2 == 0) …

  - if (percent < 0) || (percent > 100) …

Il manque les parenthèses pour encadrer la condition : if ((percent < 0) || (percent > 100)) …

  - ```
    if (minute == 60) ;
      minute = 0 ;
    ```

Il n'y a pas d'erreur de syntaxe ici, il s'agit d'une erreur de logique. En effet, le point-virgule de la première ligne fait que la seconde sera exécuté quelle que soit la valeur de *minute*.

```
if (minute == 60)
  minute = 0 ;
```

  - ```
    for (int i=0 , i < 10, i++)
      Console.WriteLine(i) ;
    ```

La syntaxe du *for* implique l'utilisation de point-virgule pour séparer les 3 éléments qui le compose :
for (int i=0 ; i < 10 ;i++) …

  - ```
    int i = 0;
    while(i < 10)
      Console.WriteLine(i);
    ```

Il n'y a pas d'erreur de syntaxe ici, il s'agit d'une erreur de logique. En effet, la variable n'est jamais modifiée dans la boucle, engendrant une boucle infinie qui affichera toujours la valeur 0.

```
int i = 0 ;
while(i < 10)
  Console.WriteLine(i++) ;
```

  - ```
    for (int i=0; i >= 10; i++)
      Console.WriteLine(i);
    ```

Il n'y a pas d'erreur de syntaxe ici, il s'agit d'une erreur de logique. En effet, le test mis en œuvre dans le *for*, implique que le corps de la boucle s'exécute tant que la variable est supérieure ou égale
à 10, ce qui est faux dès le départ. Il n'y aura donc aucune exécution du *Console.WriteLine*.

- do
  …
  string line = Console.ReadLine();
  guess = int.Parse(line);
  while (guess != answer);

La syntaxe du *do…while* impose l'utilisation d'accolades pour délimiter le corps de la boucle.

```
do{
  …
}while(guess != answer);
```

- int[] array;
  array = {0, 2, 4, 6};

L'initialisation en bloc des valeurs d'un tableau ne peut s'effectuer que lors de sa déclaration.

```
int[] array = {0, 2, 4, 6} ;
```

- int[] array;
  Console.WriteLine(array[0]);

Le tableau est declaré mais non initialisé. Il faut donc envisager l'utilisation du mot clé *new* ou d'une liste de valeur.

- int[] array = new int[3];
  Console.WriteLine(array[3]);

Le tableau créé ne contient que 3 éléments avec des indices allant de 0 à 2.

- int[] array = new int[];

Il n'est pas possible de créer un tableau sans lui affecter une taille.

- int[] array = new int[3]{0, 1, 2, 3};

La taille indiquée entre crochet ne correspond pas au nombre d'éléments fournis dans la liste.

- Module 8

Exercice 1

- **Trouver les erreurs de syntaxe dans le code suivant :**

```
interface IToken
{
        string Name( );
        int LineNumber( ) { return 42; }
        string name;
}
class Token
{
        string IToken.Name( ) { .... }
        static void Main( )
        {
                IToken t = new IToken( );
        }
}
```

C# 7

a. Une interface ne peut avoir de code, ni de déclarations de membres.
```
interface IToken{
  string Name() ;
  int LineNumber() ;
}
```
b. Aucune indication d'implémentation de l'interface.
```
class Token : IToken
```
c. La méthode *IToken.LineNumber* n'est pas implémentée
d. Il n'est pas possible d'instancier une interface.
```
IToken t = new Token() ;
```

C# 8

La possibilité d'avoir une implémentation par défaut dans une interface permet de laisser le code de *LineNumber* dans l'interface et de ne pas l'implémenter dans la classe *Token*.

- **Quel est le résultat d'exécution du code suivant :**

=> **DDBB**

d.M() => D (méthode de la classe D)
c.M() => D (*c'est* en fait *d*, par polymorphisme, la méthode appelée est celle de D)
b.M() =>B (la méthode M() de C masque celle de B, or même si *b* est *c*, ici la méthode appelée est celle de B)
a.M() =>B (*a* est en fait *b*, par polymorphisme, la méthode appelée est celle de B)

- Module 10

Exercice 3

- **Trouvez les erreurs de syntaxe dans les déclarations suivantes :**

  - public bool operator != (Time t1, Time t2) { ... }

  Les opérateurs doivent être statiques.

  - public static operator float(Time t1) { … }

  L'indication *implicit/explicit* est absente. Ici nous pourrions envisager une conversion *implicit*.

  - public static Time operator += (Time t1, Time t2) { ... }

  Il n'est pas possible de surcharger le +=.

  - public static bool Equals (Object obj) { ... }

  La méthode *Equals* est une méthode d'instance, et si nous souhaitons la redéfinir, il faudrait indiquer le mot clé *override* pour préserver le polymorphisme avec la méthode *Equals*
  de la classe *Object*.

  - public static int operator implicit(Time t1) { ...}

  Les mots clés *int* et *implicit* ont été inversés.

MS860
Annexes

Module 13 – Exemple de collection simple

```
class DoubleEndedQueue<T> : ICollection<T>, IList<T>
    {
        // Define a List<T> object to store items that are added
        // to the collection.
        private List<T> items;
        // Add a constructor to the class.
        public DoubleEndedQueue()
        {
            items = new List<T>();
        }

        #region Methods for enqueuing and dequeuing items.
        public void EnqueueItemAtStart(T item)
        {
            items.Insert(0, item);
        }

        public T DeQueueItemFromStart()
        {
            T item = items[0];
            items.RemoveAt(0);
            return item;
        }

        public void EnqueueItemAtEnd(T item)
        {
            items.Add(item);
        }

        public T DeQueueItemFromEnd()
        {
            T item = items[items.Count - 1];
            items.RemoveAt(items.Count - 1);
            return item;
        }
        #endregion

        #region ICollection<T> Members

        // Define an Add method that adds an item to the collection.
        public void Add(T item)
        {
            items.Add(item);
        }

        // Define a Clear method that removes all items from
        // the collection.
        public void Clear()
        {
            items.Clear();
        }

        // Define a Contains method that returns a bool object
        // according to whether the collection contains a particular item.
        public bool Contains(T item)
        {
            return items.Contains(item);
        }
```

```
// Define a CopyTo method that copies the entire contents of the
// collection to an array. The array is provided as a parameter,
// and the first item from the collection should be stored in the
// index that is passed as a parameter to the method.

public void CopyTo(T[] array, int arrayIndex)
{
    items.CopyTo(array, arrayIndex);
}

// Define a read-only Count property that returns the number of
// items in the collection.

public int Count
{
    get { return items.Count; }
}

// Define a read-only IsReadOnly property that returns whether
// the collection is read-only.

public bool IsReadOnly
{
    get { return false; }
}

// Define a Remove method that removes the specified item from
// the collection and returns a bool object to indicate whether
// the item was removed successfully.

public bool Remove(T item)
{
    return items.Remove(item);
}

#endregion

#region IEnumerable<T> Members

// Define a GetEnumerator method, which returns an IEnumerator<T>
// object.

public IEnumerator<T> GetEnumerator()
{
    return items.GetEnumerator();
}
#endregion

#region IEnumerable Members

// Add an IEnumerable.GetEnumerator method. This is a nongeneric
// version of the GetEnumerator method.

IEnumerator IEnumerable.GetEnumerator()
{
    // Always return the result of calling the GetEnumerator
    // method.
    return GetEnumerator();
}
```

```
        #endregion

        #region IList<T> Members

        // Add an IndexOf method that returns the index of the first
        // occurrence of an item in the collection.

        public int IndexOf(T item)
        {
            return items.IndexOf(item);
        }

        // Add an Insert method that adds an item to the collection at a
        // specified index.

        public void Insert(int index, T item)
        {
            items.Insert(index, item);
        }

        // Add a RemoveAt method that removes an item from the collection
        // at the specified index.

        public void RemoveAt(int index)
        {
            items.RemoveAt(index);
        }

        // Add an indexer that enables read/write access to items in the
        // collection, based on the specified index.

        public T this[int index]
        {
            get
            {
                return items[index];
            }
            set
            {
                items[index] = value;
            }
        }
        #endregion
    }
```