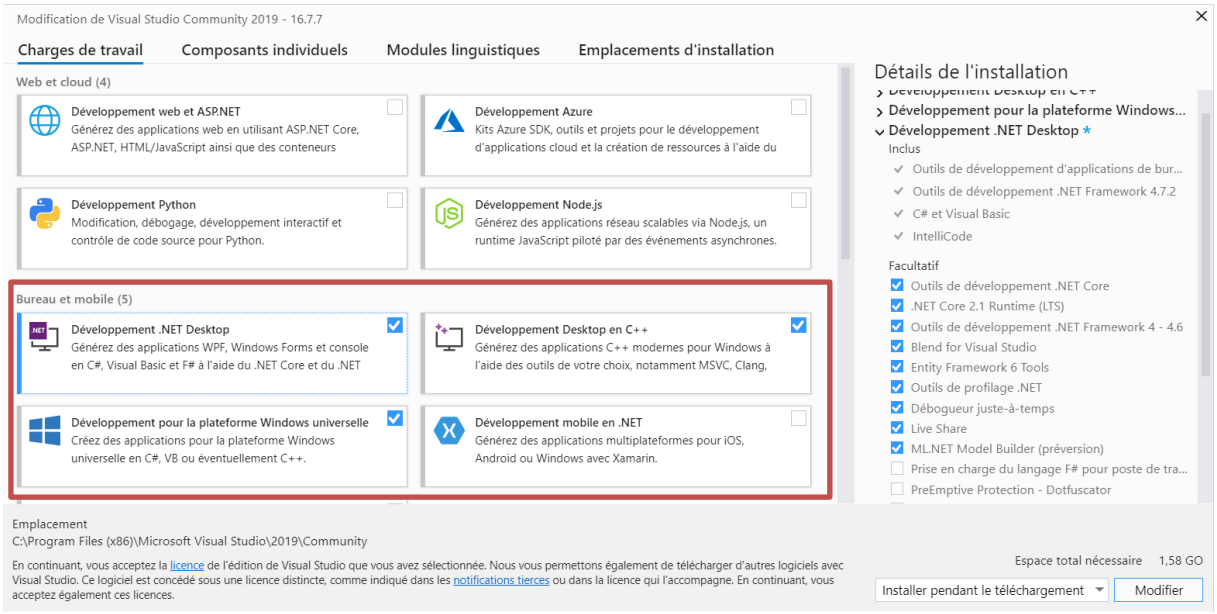
Laboratoire #1 : Introduction

Objectifs

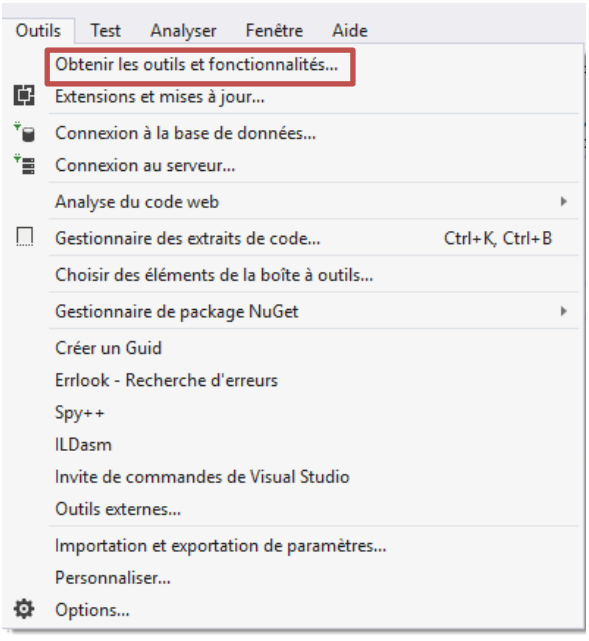
- Prise en main de VS2019 pour des applications utilisant le langage C#
- Rappel des concepts de base de l'orienté objet (types primitifs, classe et objet, variables d'instance, variables de classe, constructeurs, accesseurs, modificateurs…)
- Définition guidée d'une classe en C#
- Conventions syntaxiques propres au C#
- Manipulation des tableaux

Configuration de Visual Studio 2019

Lors de l'installation de VS2019, il faut sélectionner les 3 éléments encadrés ci-dessous.



Si vous avez déjà installé VS2019 sans avoir sélectionné « Développement .NET Desktop », il vous suffit de passer par le menu « Outils » et de sélectionner « Obtenir les outils et fonctionnalités… » comme indiqué ci-dessous.



Vous pourrez ainsi modifier la configuration de votre version de VS2019.

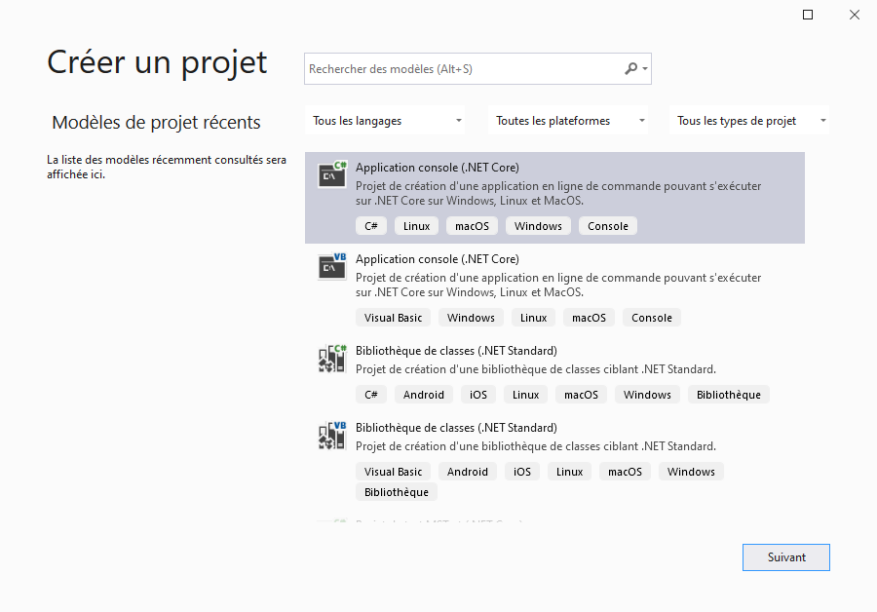
Vous pouvez, via le même outil, ajouter une langue et, par la suite, en changer via « Outils » → « Options… », dans « Environnement », dans « Paramètres internationaux ».

Exercice 1 : définition guidée d'une classe

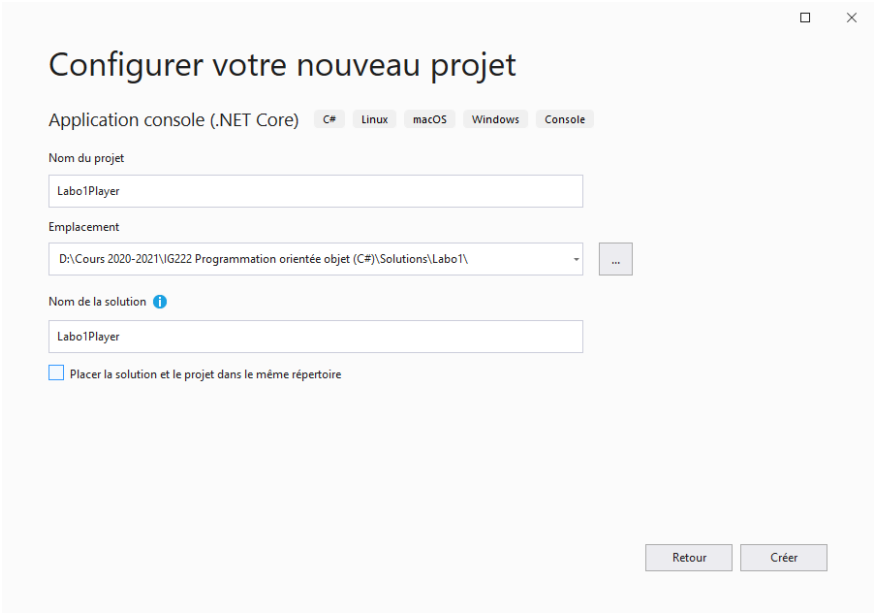
Étape 1 : prise en main de l'IDE – création d'une solution

Lancez le logiciel d'environnement de développement qu'est Visual Studio version 2019. Vous pouvez passer soit par la « Page d'accueil » soit par le menu « Fichier » …

Via le menu « Nouveau Projet », créez une solution en utilisant un *template* déjà installé qui permet de générer une solution comprenant une application console (cf. images ci-dessous).

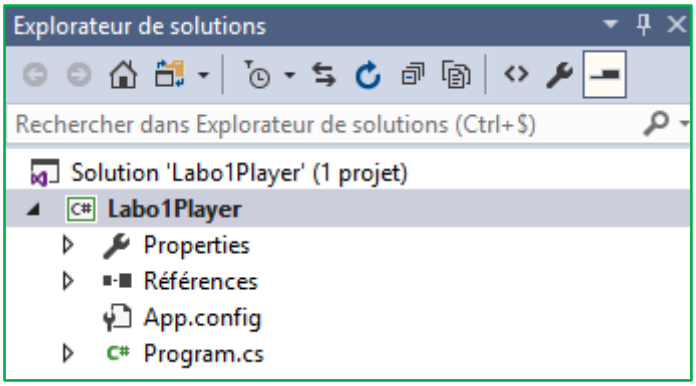


Une solution est un ensemble de projets qui peuvent être écrits dans des langages distincts (C#, VisualBasic, C++…). Le nom du projet (et de la solution) est Labo1Player. Vérifiez que le chemin d'accès est celui souhaité !



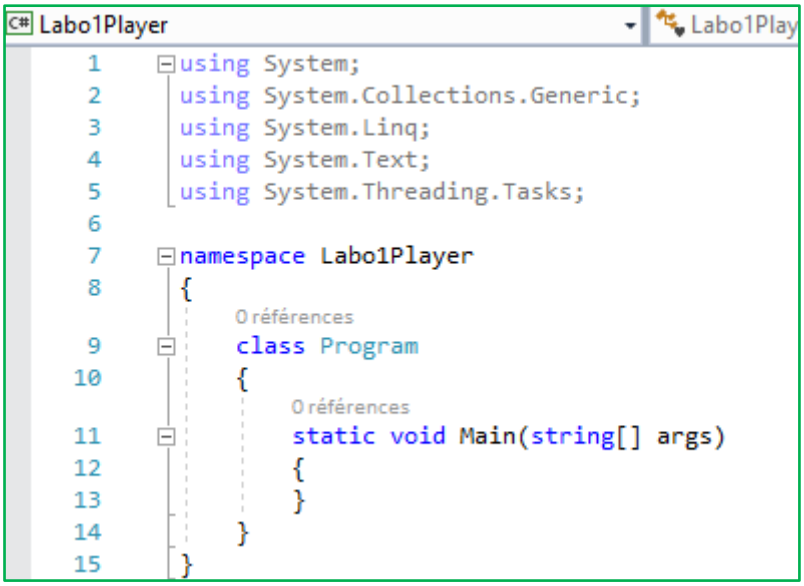
Vous arrivez à un écran composé de plusieurs parties, comme lors du cours de Langage de programmation avancé de 1IG.

Dans la fenêtre « Explorateur de solutions », vous voyez le nom de la solution ainsi que le premier projet inclus dans celle-ci.



Par défaut, votre projet contient un fichier appelé *Program.cs* où se trouve la description de la classe `Program`. Cette dernière est la classe où vous pourrez créer des objets, tester des appels de méthodes…

Si vous cliquez sur *Program.cs*, le contenu du fichier est affiché dans la partie « principale » de l'interface :



La méthode principale `Main` est le point d'entrée de votre application.

En C#, conventionnellement, les **noms des méthodes commencent par une majuscule**.

Comparatif avec Java

En Java, une méthode commence par une minuscule. En dehors de cette nuance, la signature des méthodes est similaire dans les deux langages.

Dans cet exemple, le fichier qui contient la classe `Program` porte le même nom que celle-ci, mais ce n'est pas une obligation. Les règles relatives au nom des méthodes et des classes sont plus libres en C# qu'en Java.

Rappel de Java

À quelques exceptions près, en Java, chaque classe doit être décrite dans un fichier séparé portant le nom de la classe.

Un fichier ne peut donc généralement ne contenir qu'une classe et les noms doivent correspondre.

En C#, ces deux contraintes sont levées : (1) on peut placer plusieurs classes dans le même fichier ; (2) il n'y a aucune règle liant le nom des classes et le nom des fichiers ; et (3) on peut même répartir la définition d'une classe sur plusieurs fichiers.

Toutefois, il est important de noter que c'est une bonne idée ("de bonnes pratiques") de continuer à respecter les règles du Java même si celles-ci ne sont pas imposées en C#. (Lire l'encadré suivant pour une discussion plus précise)

Pragmatique / en pratique

Mettre plusieurs classes dans un même fichier. Il est déconseillé de placer plusieurs classes dans un même fichier pour des raisons de structuration, de visualisation globale d'un projet et de maintenabilité par la suite, exceptés dans certains cas non abordés au bloc 2.

Répartir la définition d'une classe entre plusieurs fichiers. Cette façon de faire n'est acceptable que dans deux cas : (1) quand le code est généré automatiquement par le *framework* .NET lors de la construction de projets avec interfaces graphiques (voir exemples au cours de théorie), et (2) quand, dans le milieu professionnel, la rédaction du code concernant une même classe est réalisée par deux équipes différentes (une équipe fournit le début de la classe à l'autre équipe dans un premier fichier, la seconde équipe complète cette classe sans modifier à ce qui lui a été fourni et agit donc dans un second fichier).

Étape 2 : ajout de la classe dans le projet

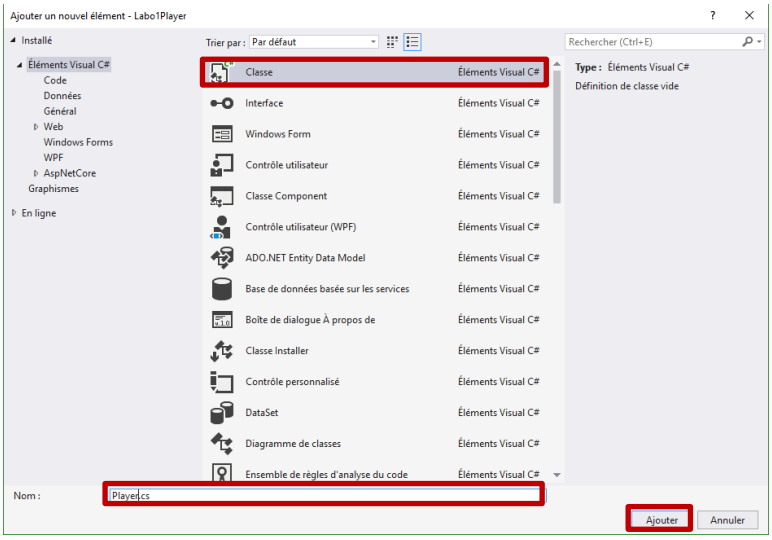
Ajoutez un nouveau fichier *Player.cs* définissant la classe `Player`.

Plusieurs manières de procéder s'offrent à vous, dont les deux suivantes :

- si vous vous placez dans « l'Explorateur de solutions », avec un clic droit sur le projet, vous pouvez lui ajouter un nouvel élément via le sous-menu « Ajouter » puis « Nouvel élément… » ;

- si vous passez par le menu « Projet », après avoir sélectionné le projet dans « l'Explorateur de solutions », vous pouvez choisir le sous-menu « Ajouter une classe… ».

Vous devez ajouter une nouvelle classe, donc sélectionnez « Class » et nommez-la *Player.cs*.



Ce fichier comprend la classe `Player` dont une partie du code est généré automatiquement.

Le nom d'une **classe** commence par une **majuscule**, ne peut comprendre que des caractères alphanumériques (lettres accentuées ou non) ou le caractère _ à condition de ne pas commencer par un chiffre.

Par convention, le nommage respecte le style « UpperCamelCase » (aussi appelé « PascalCase »).

Comparatif avec Java

En Java, un nom de classe commence également par une majuscule et respecte la même convention de style.

Observez les divers éléments qui composent le code généré pour la classe `Player`. Pour chacun de ces éléments, tentez de vous rappeler l'équivalent en Java et identifiez les points communs et les différences entre Java et C#.

Les premières lignes du fichier méritent déjà un commentaire :

Namespace

Un *namespace* ou espace de noms est une structure hiérarchique groupant des classes par utilité ou par contexte d'utilisation.
C'est avant tout une organisation logique, mais la plupart du temps, l'organisation physique (les dossiers et les fichiers qui s'y trouvent) est semblable à la découpe en *namespaces*.
On peut renommer un *namespace* (dans le milieu professionnel, on ajoute souvent le nom de l'auteur, du projet et/ou de l'éditeur du logiciel tout au début du nom) mais attention à préserver la cohérence. Voici deux exemples :

- *Henallux.IG.Presences.UI* est le *namespace* qui contient les définitions de types de l'interface utilisateur du projet Presences, développé par la section IG de l'Henallux.
- *Hennalux.IG.Presences.DAO* contient la définition de la couche d'accès aux données du même projet (DAO pour Data Access Objects).

Using

C'est un mot clé permettant de préciser que la classe utilise une librairie déjà existante. On importe ainsi les *namespaces* contenant ces librairies. Ce mot-clé sert aussi à autre chose, mais ce n'est pas l'objet de ce labo.

Par défaut, Visual Studio ajoute plusieurs importations de *namespaces* au début du fichier. Celles-ci correspondent aux bibliothèques standards utilisées par la majorité des programmes.

Comparatif avec Java

Namespaces et packages. La notion de *namespace* en C# correspond plus ou moins à celle de *package* en Java. En Java, chaque *package* doit correspondre à une structure physique (un dossier) dans lequel les fichiers des classes (ou des sous-packages) se trouvent.

Ce n'est pas le cas en C#, où le *namespace* est une organisation purement logique : cela signifie que, dans l'esprit des concepteurs (= au niveau logique), les classes d'un même *namespace* sont apparentées, mais elles ne doivent pas forcément correspondre à des fichiers « physiquement » situés dans un même répertoire (on peut toutefois le faire, pour plus de clarté).

Using et import. Le mot *using* de C# correspond à *import* en Java quand on souhaite utiliser des éléments d'un *package*. Une différence notable : *using* porte sur un *namespace* alors que *import* cible une (ou plusieurs) classe(s).

Étape 3 : présentation de la classe Player

Soit le schéma suivant correspondant à une classe `Player`[1].

Player
- firstName : string - lastName : string - birthday : DateTime - skillRating : int - sponsored : bool

[1] Ne foncez pas ! Veuillez suivre les démarches et explications indiquées.

Dans un jeu tel qu'Overwatch, un joueur est notamment caractérisé par son nom (`lastName`), son prénom (`firstName`), sa date de naissance (`birthday`) et un score (`skillRating`).

Le score est un entier entre 1 et 5000 si le joueur accepte de participer au jeu de manière compétitive.

En analysant les attributs d'un peu plus près, vous constatez qu'il n'y a pas de variable d'instance booléenne pour indiquer si le joueur participe ou non à la compétition. Cette information est mémorisée via le score (`skillRating`) qui vaut `0` si le joueur n'est pas compétiteur et qui est initialisé à `1` s'il est compétiteur.

Enfin, on mémorise le fait que le joueur est sponsorisé ou pas au moyen d'un attribut de type booléen `sponsored`.

Étape 4 : rappel d'orienté objet et présentation des types en C#

Cette étape regroupe quelques rappels sur l'orienté objet ainsi que sur les types en C#. En première lecture, vous pouvez éventuellement ne pas lire ce qui suit en détail, mais n'oubliez pas d'y revenir par la suite !

L'orienté objet…

Un objet est identifié par son nom, son état et son comportement. L'état est l'ensemble des valeurs données aux attributs de la classe dont il est une instance et ce, à un moment donné. L'état peut évoluer au cours du temps.

L'encapsulation est le fait de cacher aux yeux de l'extérieur une partie de l'objet déclarée privée, et de ne la rendre accessible, si c'est souhaité, uniquement via des méthodes publiques.

Tout l'état ou une partie de l'état peut être accessible via des méthodes appelées accesseurs (ou *getters*) ; tout l'état ou une partie de l'état peut être modifié par des méthodes appelées modificateurs (ou *setters*).

Avant d'ajouter des attributs à la classe `Player`, lisez ce qui suit !

Les types standards (entiers, booléens, réels)

Le langage C# est sensible à la casse. Par convention, un attribut commence par une minuscule. Un nom de variable ne peut dépasser 511 caractères. Il ne peut pas commencer par un chiffre ; il peut comporter des caractères alphanumériques, lettres accentuées ou non, le ç, le µ et le _.[2]

Le framework .NET comprend beaucoup de types prédéfinis dont une quinzaine appelés types intégrés ou primitifs tels que `Int32`, `UInt64`, `Double`, `Decimal`, `Char`, `String` (jusque 4 Go de caractères) et `Boolean`. Plutôt que d'utiliser les noms rébarbatifs de ces types, on emploie généralement les alias (ou synonymes) mis en place, à savoir : `int`, `ulong`, `double`, `decimal`, `char`, `string` et `bool`.

En pratique, utilisez `int` pour les entiers et `double` pour les réels ; dans le cas de sommes d'argent, de vitesses ou d'autres nombres décimaux dont la partie décimale doit être

[2] Il peut aussi comprendre un mot réservé du langage à condition de le précéder de @ ou de tout autre caractère permis.

mémorisée précisément, il vaut mieux utiliser le type `decimal` (qui correspond à une représentation des chiffres du nombre plutôt qu'à une approximation de sa valeur en binaire).

Par défaut, les attributs de type standard sont initialisés au zéro du type.

Type de valeurs standards et comparatif avec Java

On a employé ci-dessus le terme « type » pour parler de Int32, UInt64 et des autres noms associés aux valeurs standards. Si on n'a pas utilisé le mot « classe », c'est parce ce qu'il ne s'agit pas de classes à proprement parler. En C#, deux grandes catégories coexistent : les classes (permettant de construire des objets) et les structures (dont les valeurs sont générées en mémoire de manière différente des objets). Le terme « type » est une appellation générique pour les classes et les structures. Int32, UInt64 et les autres noms cités ci-dessus sont en fait des structures.

En Java, il n'existe que des classes et des types de bases prédéfinis tels que int, double et bool. En C#, les syntaxes int et Int32 sont des synonymes, désignant toutes les deux la même structure. En Java, le type de base int et la classe Integer sont bien distinctes… mais il existe des mécanismes automatiques permettant de convertir les int en Integer et les Integer en int chaque fois que c'est nécessaire. Ces conversions portent le nom de « *boxing* » (mise en boîte, type simple vers classe) et « *unboxing* » (déballage, classe vers type simple) ; on parle aussi de « l'*autoboxing* » pour évoquer le fait que ces transformations se font automatiquement.

Les chaines de caractères

Pour les chaines de caractères, le C# offre deux syntaxes : `string` ou `String`, les deux étant des synonymes désignant la classe `String` décrite dans le *namespace* `System`.

La comparaison des chaines caractères se fait par égalité (==) et non via `equals`.

Comparatif avec Java

En Java, il faut impérativement comparer des chaînes de caractères en utilisant une méthode telle que `equals` ; le test `str1 == str2` quant à lui compare les références.

Lors de la définition d'une classe, le langage C# permet non seulement de définir des méthodes portant des noms usuels (comme toString ou getValue) mais aussi de (re)définir la signification de certains symboles d'opérations comme « + », « * » ou encore « == ».

Dans le cas des chaînes de caractères, par exemple, l'opérateur « == » a été redéfini pour comparer le contenu plutôt que simplement l'adresse, ce qui explique pourquoi cette écriture fonctionne.

Par défaut, les variables de type `string` sont initialisées à `null`.

Étape 5 : attributs et constructeurs dans Player

Dans un premier temps, ajoutez les attributs `firstName`, `lastName`, `birthday`, `skillRating` sans indiquer `private` ou autre indicateur de visibilité (la syntaxe est la même qu'en Java).

Pour la date de naissance, vous pouvez utiliser la classe prédéfinie `DateTime`, dont un des constructeurs reçoit comme argument (dans l'ordre) l'année, le numéro du mois et le numéro du jour.

Créez un constructeur général sachant que prénom, nom et date de naissance portent les noms `firstName`, `lastName` et `birthday` dans les paramètres d'entrée. Le 4e paramètre sera un booléen `isRanked` indiquant si le joueur joue en compétitif ou pas (et déterminant ainsi son `skillRating` initial).

Notez que la classe ne comporte aucun attribut « isRanked ». Un constructeur peut tout à fait recevoir un ou plusieurs arguments qui ne correspondent pas directement à des attributs mémorisés. Il n'y a aucune restriction à ce sujet.

Comparatif avec Java

Les constructeurs s'écrivent de la même manière en Java et en C#. La manière d'affecter les paramètres formels aux variables d'instance (création de l'état à un moment donné) est également identique.

Les variables d'instance sont garnies avec les paramètres via, pour l'instant,

```
this.lastName = lastName;
```

où le membre de droite de l'instruction est le paramètre et le membre de gauche fait référence à l'attribut.

Avez-vous pensé à utiliser l'opérateur conditionnel ternaire pour initialiser `skillRating`. Si oui, continuez ; si non, modifiez votre code.

Créez un autre constructeur plus particulier pour les compétiteurs où on ne renseigne que le prénom, le nom et la date de naissance.

Note. Pour le moment, implémentez les deux constructeurs séparément plutôt que de faire en sorte que le second appelle le premier.

Testez votre code en créant deux joueurs dans la classe `Program`, un compétiteur et un autre. Essayez d'imprimer le prénom d'un des joueurs grâce au code suivant :

```
Console.WriteLine(…);    // impression avec passage à la ligne
Console.ReadKey();       // lit un caractère (cf. getchar en C)
```

Vous n'avez pas accès au prénom ! En effet, par défaut, les attributs (et les méthodes) ne sont pas publics. Rendez l'attribut `public` ; tout fonctionne !

Nous perfectionnerons la visibilité des attributs plus loin, et étudierons toutes les nuances de protection dans un autre labo.

Étape 6 : accesseurs et modificateurs

Protégez au maximum les attributs en leur imposant la visibilité `private` (c'est la visibilité par défaut des attributs et méthodes d'une classe en C# mais autant le spécifier explicitement).

Pour pouvoir accéder au contenu des attributs privés, il faut mettre en place des *getters* et des *setters*.

Notez que tous les attributs n'auront pas forcément un modificateur (*setter*) ou et un accesseur (*getter*) : un programmeur prudent réfléchit à ce qu'il veut rendre accessible et pour qui.

Les nom et prénom du joueur doivent être accessibles ensemble ; ce qui donne la méthode `GetName` (attention, par convention, on met une majuscule au nom de méthode) :

```
public string GetName()
{
  return this.lastName + " " + this.firstName;
}
```

Créez les 3 *getters*/*setters* :

Date de naissance. Créez une méthode qui rend accessible la date de naissance sous forme de chaîne de caractères jj/mm/aaaa. Si `date` est un objet de type `DateTime`, vous pouvez obtenir respectivement le numéro du jour, du mois et de l'année via `date.Day`, `date.Month` et `date.Year` (notez les majuscules ; il s'agit de propriétés, un sujet abordé dans un futur laboratoire).

Compétiteur (*setter*). Un joueur non compétiteur peut vouloir le devenir. Écrivez le modificateur (`SetRanked`) correspondant en vérifiant que le joueur ne l'était pas déjà ! S'il l'est déjà, on ne fait rien (dans un programme finalisé, un message d'erreur apparaîtrait).

Compétiteur (*getter*). Créez une méthode `IsRanked` qui renvoie un booléen indiquant si le joueur est compétiteur. Elle ne sera accessible qu'au sein de la classe `Player`. Ensuite, modifiez la méthode `SetRanked` pour qu'elle utilise `IsRanked`.

Testez votre programme avant d'aller plus loin.

Étape 7 : appels entre constructeurs

Quand, dans une classe, un constructeur général a été écrit, les autres devraient être écrits en fonction d'un appel à ce constructeur général afin d'éviter de recopier du code... Principe du « point de modification unique » !

Réécrivons donc le second constructeur de manière à ce qu'il fasse appel au premier.

Comparatif avec Java

En Java, un constructeur peut contenir un appel à un autre constructeur, sous la forme `this(…)`, mais celui-ci doit être la toute première instruction du code.

Le langage C# met en évidence cette restriction en plaçant l'appel sur une ligne à part, hors des accolades.

```
public Player(string firstName, string lastName, DateTime birthday) :
this(lastName, firstName, birthday, true)
{ }
```

Testez votre code.

Étape 8 : méthode ToString

La méthode `ToString` doit produire l'affichage suivant :

> Nom et prénom : …
> Date de naissance : jj/mm/aaaa
> Compétiteur ou Non compétiteur

La méthode ToString par défaut en C# est la suivante :

```
public override string ToString()
{
     return base.ToString();
}
```

Comparatif avec Java

L'appel `base.ToString()` correspond à `super.ToString()` en Java.

Notez la différence de mots-clés, `base` en C#, `super` en Java.

Remplacez le corps de la méthode par :

```
public override string ToString()
{
    string output = "";
    //À vous de compléter
    return output;
}
```

À vous de compléter la méthode afin d'obtenir le résultat attendu. Prenez soin d'utiliser les méthodes déjà écrites.

Étape 9 : une autre méthode

Les points d'un compétiteur évoluent au cours du temps. Écrivez la méthode `ModifySkillRating` qui modifie les points du joueur en respectant les consignes suivantes. Avant de coder, imaginez un jeu de tests permettant de vérifier votre futur code ; combien de cas de tests envisageriez-vous ?

a. Le nombre de points (à additionner/à retirer) est un argument d'entrée (si c'est à retirer, les points sont négatifs).
b. Aucun traitement n'est fait si le joueur n'est pas en compétition.
c. Si les points ajustés dépassent 5000, le nombre de points du joueur vaut 5000 maximum.
d. Si les points ajustés sont inférieurs à 1, ils sont égalisés à 1.

Pragmatique / en pratique

Visual Studio offre la possibilité de définir des régions dans le code, comme par exemple : la région des variables d'instance, la région des constructeurs…

Pour ce faire, voici la syntaxe à utiliser (dans le cas des variables d'instance) : on écrit #region suivit du libellé de la région au début et #endregion à la fin :

```
#region attributes
public string firstName;
string lastName;
DateTime birthDay;
int skillRating;
#endregion
```

En cliquant sur le symbole à gauche de la région, on a : attributes

Notez qu'il s'agit d'une spécificité de l'IDE Visual Studio plutôt que du langage C#. Ces ajouts n'ont aucun effet sur le code si ce n'est qu'ils peuvent le rendre plus lisible en permettant de cacher/fermer ou de révéler/ouvrir au gré les régions.

Définissez une région pour les constructeurs et réduisez les deux régions afin d'avoir :

attributes

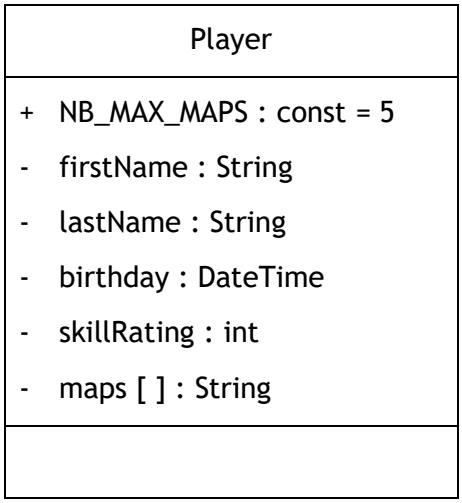
constructors

Exercice 2 : gestion des cartes

Les cartes/décors que le joueur a utilisées au cours de ses dernières parties (une carte par partie) sont également mémorisées. Pour cela, on utilise un tableau qui contient le libellé des 5 dernières cartes utilisées. On ne retient pas plusieurs fois la même carte. Il se peut aussi qu'il y ait moins de 5 cartes différentes ; par exemple si le joueur utilise toujours les 3 mêmes cartes, seules ces cartes sont mémorisées.

Deux éléments sont donc ajoutés à la classe Player : le nombre maximum de cartes, `NB_MAX_MAPS`, ainsi que le tableau des cartes, `maps`.

Voici le schéma UML mis à jour :



Étape 1 : création d'un tableau

Le tableau stocke les libellés des dernières cartes utilisées par le joueur, au maximum `NB_MAX_MAPS` ; s'il y en a moins, les dernières cellules sont à `null`. Les libellés vont de la plus récente à la plus ancienne, la dernière carte utilisée se trouvant toujours en tête du tableau.

Voici un exemple. Supposons, pour cet exemple, qu'on décide de conserver le nom des 5 dernières cartes utilisées. Initialement, lors de la création du joueur, le tableau contiendra les informations suivantes. N'oubliez pas que la valeur `null` est la valeur par défaut que le langage C# utilise pour initialiser les cellules.

<null>	<null>	<null>	<null>	<null>

Si le joueur fait une première partie en utilisant la carte « Canyon », le tableau devient

Canyon	<null>	<null>	<null>	<null>

Si, ensuite, il joue plusieurs parties en utilisant successivement les cartes « Forteresse » et « Forêt », on obtiendra

Forêt	Forteresse	Canyon	<null>	<null>

Si le joueur se lance ensuite dans une partie utilisant à nouveau la carte « Forteresse », celle-ci devient la carte la plus récemment utilisée et passe au début du tableau.

Forteresse	Forêt	Canyon	<null>	<null>

S'il participe ensuite à des parties utilisant les cartes " Désert " puis " Oasis ", le tableau devient

Oasis	Désert	Forteresse	Forêt	Canyon

Si, à ce moment-là, le joueur refait une partie sur la carte « Forêt », à nouveau, un échange s'opère et « Forêt » passe en tête.

Forêt	Oasis	Désert	Forteresse	Canyon

Finalement, s'il joue ensuite une partie sur la toute nouvelle carte « Cavernes », voici ce que devient le tableau :

Cavernes	Forêt	Oasis	Désert	Forteresse

Déclarez une constante `NB_MAX_CARDS` valant 5.

```
public const int NB_MAX_MAPS = 5;
```

Comparatif avec Java

Le mot réservé `const` du C# indique une constante qui doit être initialisée lors de sa déclaration. La valeur d'initialisation doit être calculable à la compilation, et est donc la même pour toutes les instances de la classe. (Ainsi, chaque constante est automatiquement « de classe », le mot-réservé `static` étant inutile et interdit dans une déclaration de constante.)

En Java, cela correspond plus ou moins au mot réservé `final`, à l'importante différence suivante près : une variable déclarée `final` ne doit pas obligatoirement être initialisée lors de sa déclaration ; elle peut l'être plus tard, dans un constructeur par exemple. Cela signifie que la valeur peut être différente pour chaque instance.

Déclarez votre tableau `maps` comme en Java en l'initialisant à la bonne taille.

Étape 2 : ajout de fonctions manipulant le tableau

Au départ, le joueur n'a pas choisi de carte. Le constructeur ne doit donc pas être modifié.

Prévoyez une méthode `AddMap` suivant ce qui a été décrit à la page précédente.

Prévoyez une méthode `ListingMaps` qui renvoie la liste des libellés sous la forme :

1 - Le canyon désertique
2 - La forêt enchantée
3 - USS Entreprise
4 -
5 -

Revoyez votre méthode `ToString` pour qu'elle affiche aussi la liste des cartes. Si le joueur a utilisé moins de 5 cartes, rien n'est affiché en lieu et place du nom de la carte.

Testez votre code.

Étape 3 : parcourir un tableau - foreach

Pour parcourir un tableau, vous pouvez soit employer un indice et l'instruction `while`/`for` soit utiliser l'instruction `foreach`.

La syntaxe est :

```
foreach (int val in tabEntiers) { ... }
```

Comparatif avec Java et Javascript


En Javascript, la syntaxe est :

```
for (let val of tabEntiers) { ... }
```

Ne confondez pas avec `for (let val in tabEntiers).`

En Java, on écrit :

```
for (int val : tabEntiers) { ... }
```

Modifiez la fonction `ListingMaps` afin d'utiliser une boucle `foreach`.

Étape 4 : test

Pour faire quelques tests, il serait intéressant de créer deux méthodes, dans la classe Program, qui feraient le travail.

> *Program testing can be used to show the presence of bugs,*
> *but never to show their absence!*
>
> *Dijkstra (1970)*

Ces deux méthodes sont définies ci-dessous. Lisez-les avant de les recopier et vérifiez que vous avez compris de quoi il s'agit !

```
public static void AssertMaps(string expected, Player p)
{
    Console.WriteLine("Expected answer : " + expected);
    Console.WriteLine(p.ListingMaps());
}
```

```
public static void TestMaps() {
    // Mise en place
    Player p = new Player("Indiana", "Jones", new DateTime(1899,7,1));
    // Test
    AssertMaps("aucune", p);
    // Mise en place
    p.AddMap("Canyon");
    // Test
    AssertMaps("Canyon", p);

    // Mise en place
    p.AddMap("Forteresse");
    p.AddMap("Forêt");
    // Test
    AssertMaps("Forêt, Forteresse, Canyon", p);

    // Mise en place
    p.AddMap("Forteresse");
    // Test
    AssertMaps("Forteresse, Forêt, Canyon", p);

    // Mise en place
    p.AddMap("Desert");
    p.AddMap("Oasis");
    // Test
    AssertMaps("Oasis, Desert, Forteresse, Forêt, Canyon", p);

    // Mise en place
    p.AddMap("Forêt");
    // Test
    AssertMaps("Forêt, Oasis, Desert, Forteresse, Canyon", p);

    // Mise en place
    p.AddMap("Caverne");
    // Test
    AssertMaps("Caverne, Forêt, Oasis, Desert, Forteresse", p);
}
```

En faisant appel à la méthode `TestMaps` dans la fonction principale, vous pourrez repérer d'éventuels bugs...

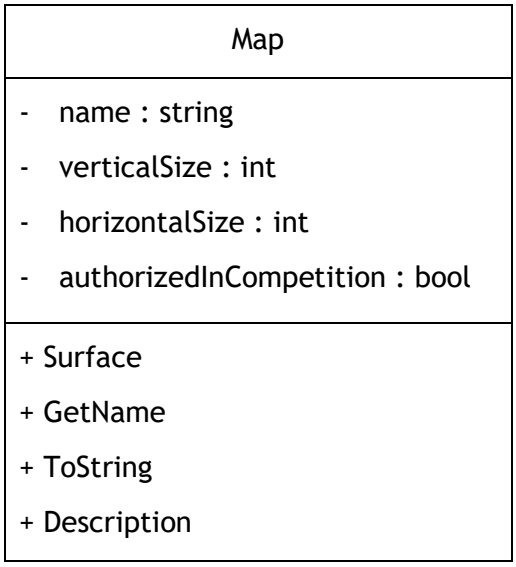
Enfin, assurez-vous que vous avez réalisé la méthode `AddMap` de l'étape 2 en ne parcourant le tableau qu'une seule fois ! Si ce n'est pas le cas, réécrivez cette méthode pour que ça le soit et recommencer les tests.

Exercice 3 : ajout d'une classe Map

Au lieu de mémoriser les cartes sous la forme d'une `string`, vous allez créer une classe pour conserver des informations concernant une carte.

Étape 1 : le schéma UML

Soit le schéma suivant correspondant à une classe `Map`.



Les tailles de la carte sont exprimées en nombres de cases.

Plusieurs constructeurs sont à prévoir :

- un constructeur standard,
- un constructeur considérant que la carte est, par défaut, autorisée en compétition,
- un constructeur considérant que la carte est carrée si seule une des deux tailles est donnée.

Définissez un accesseur `GetName` pour le nom de la carte. Il sera utilisé lors de l'étape suivante…

La méthode `Surface` renvoie la surface occupée par la carte en nombre de cases.

La méthode `ToString` renvoie une chaine de caractères composée du nom de la carte `name` et de sa taille au format suivant « (`verticalSize` x `horizontalSize`) », par exemple « *Forteresse (24 X 17)* » .

La méthode `Description` affiche la description de la carte au format suivant :

Forteresse (taille 24 x 17, surface 408 cases)
Pas utilisable / Utilisable en compétition

Faites en sorte que dans le cas où la carte n'est pas utilisable en compétition, « Pas utilisable en compétition » soit en rouge, et que dans l'autre cas, « Utilisable en compétition » soit en bleu.

Pour cela, utilisez les instructions suivantes, au bon endroit :

```
Console.ForegroundColor = ConsoleColor.Red;
Console.ForegroundColor = ConsoleColor.Blue;
```

Étape 2 : intégrer la classe Maps au reste du programme

Pour terminer, modifiez le code de la classe `Player` afin qu'il ne considère plus les cartes comme étant des `string`, mais bien des `Map` !

Utilisez à nouveau les méthodes de test pour mettre en évidence d'éventuels bugs.