12/09/2023

Programmation C#

Avec Unity – Partie 1 – FO-S008

Franck OURION

UNIVERSITE DE LORRAINE

TABLE DES MATIERES

INSTALLATION D'UNITY

L'installation de « Unity » est couverte via ce lien ou le module « Unity Learn » : Unity Essential.

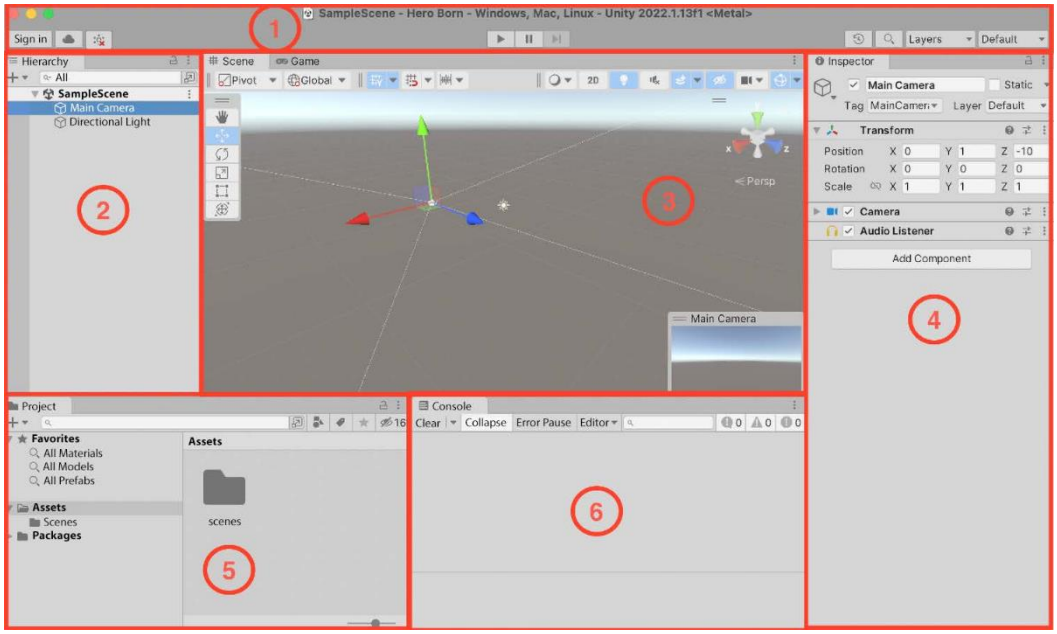
Documentation indispensable de Unity.

MANIPULATIONS DE BASE

CREATION D'UN NOUVEAU PROJET

Compétence : Get started with the Unity Hub.

NAVIGATION DANS L'EDITEUR



Nous allons donc examiner chacun de ces panneaux plus en détail :

La **BARRE D'OUTILS (1)** est la partie supérieure de l'éditeur Unity. À partir de là, vous pouvez vous connecter à un compte Unity, gérer les services, collaborer avec une équipe (groupe de boutons à l'extrême gauche), jouer et mettre en pause le jeu (boutons du centre). Le groupe de boutons le plus à droite contient une fonction de recherche, des **LAYERMASKS, etc**

La fenêtre **HIERARCHY (2)** affiche tous les éléments présents dans la scène du jeu. Dans le projet de démarrage, il ne s'agit que de la caméra par défaut et de la lumière directionnelle, mais lorsque nous créerons notre environnement prototype, cette fenêtre commencera à se remplir avec les objets que nous ajouterons dans la scène.

Les fenêtres **GAME** et **SCENE (3)** sont les aspects les plus visuels de l'éditeur. Considérez la fenêtre **SCENE** comme votre scène, où vous pouvez déplacer et arranger des objets 2D et 3D. Lorsque vous appuyez sur le bouton **Play**, la fenêtre **GAME** prend le relais et affiche la vue de la **SCENE** et toutes les interactions programmées. Vous pouvez également utiliser la vue de la **SCENE** lorsque vous êtes en mode de jeu.

La fenêtre de **L'INSPECTOR (4)** permet de visualiser et modifier les propriétés des objets de la scène. Si vous sélectionnez CAMERA PRINCIPALE dans la **HIERARCHY,** vous verrez plusieurs parties affichées, qu'Unity appelle des composants (**COMPONENTS**), tous accessibles depuis **L'INSPECTOR**.

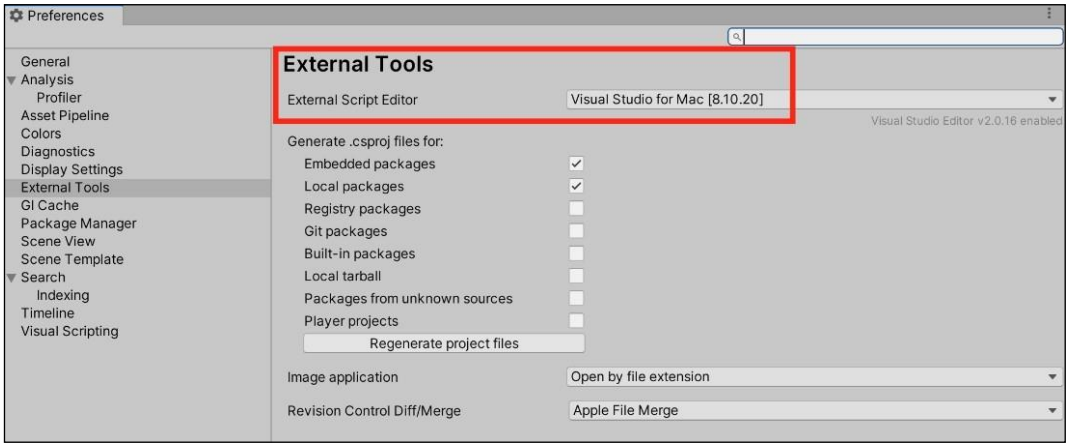
La fenêtre **PROJECT (5)** contient toutes les ressources qui se trouvent actuellement dans votre projet.

La fenêtre **CONSOLE (6)** est l'endroit où s'affiche toute sortie que nous voulons que nos scripts impriment : Cela permet de tester ou débugger le code.

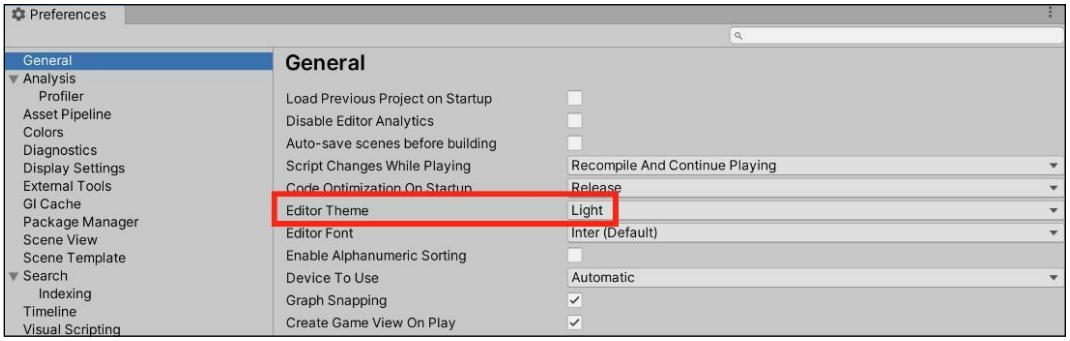
Si l'une de ces fenêtres est fermée par accident, vous pouvez la rouvrir à tout moment à partir du menu **Window>General**

Description plus détaillée des fonctionnalités.

Avant de continuer, il est important que Visual Studio soit configuré comme éditeur de script pour votre projet. Allez dans le menu **Edit>Preferences** et vérifiez que l'éditeur de script externe est réglé sur Visual Studio 2022



Enfin, si vous souhaitez passer du mode clair au mode foncé, allez dans le menu **Edit>Preferences** et modifiez



UTILISER C# AVEC UNITY

Unity est le moteur dans lequel vous allez créer des ***SCRIPTS C#*** et des ***GAMEOBJECTS***, mais la programmation proprement dite s'effectue dans un autre programme appelé Visual Studio -> **MVS**

TRAVAILLER AVEC DES SCRIPTS C#

Il existe plusieurs façons de créer des ***SCRIPTS C#*** à partir de l'éditeur :

- Sélectionnez dans le menu **Assets>Create>C# Script**
- Sous l'onglet **PROJECT**, sélectionnez l'icône **+** et choisissez **C# Script.**
- Cliquez avec le bouton droit de la souris sur le dossier **Assets** dans l'onglet **PROJECT** et sélectionnez **Create>C# Script** dans le menu contextuel.
- Sélectionnez n'importe quel objet de jeu dans la fenêtre **HIERARCHY** et cliquez le bouton en bas **sur Add a component > New script**.

> ***Il faut toujours placer les scripts dans un sous-dossier nommé Scripts du dossier Assets***

PRESENTATION DE L'EDITEUR VISUAL STUDIO

Un double – clic sur le script ouvre Visual Studio.

OUVERTURE D'UN FICHIER C#

Attention aux erreurs de dénomination :

> ***Il faudra nommer correctement tout de suite le script sous réserve de devoir renommer le nom de la classe associée au script. Utiliser C# avec Unity nous oblige à ce que le nom du fichier de script (extension .cs automatique) soit le même que le nom donné à la classe.***

SYNCHRONISATION DES FICHIERS C#

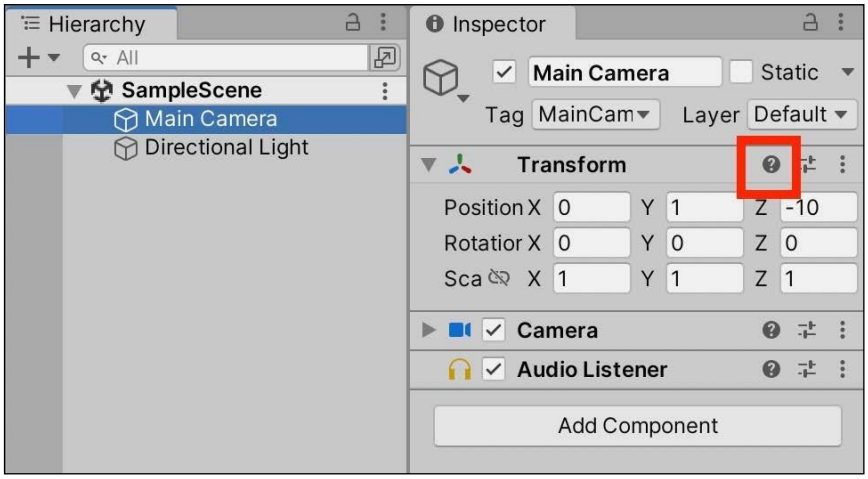
Dans le cadre de leur relation, Unity et Visual Studio communiquent entre eux pour synchroniser leur contenu. Cela signifie que si vous ajoutez, supprimez ou modifiez un fichier de script dans une application, l'autre application verra automatiquement les changements.

EXPLORER LA DOCUMENTATION

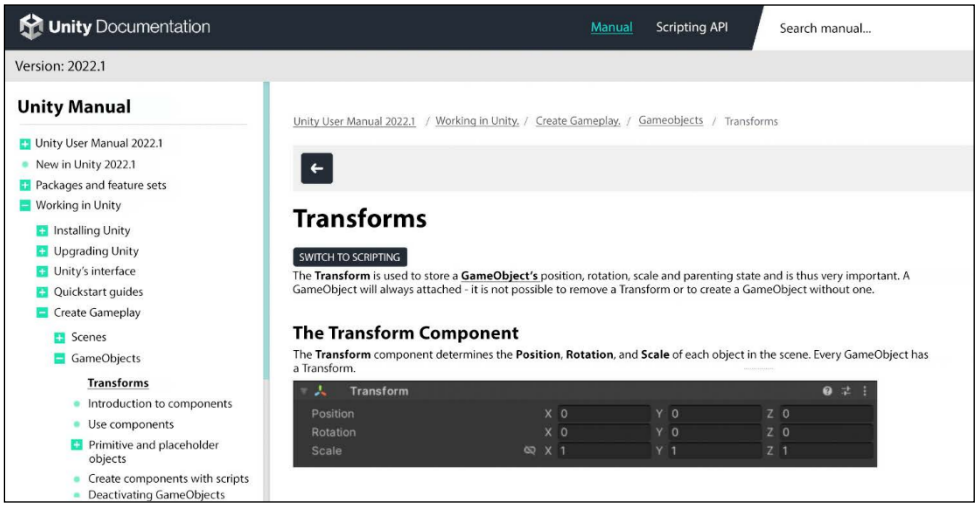
Il est important de prendre de bonnes habitudes dès le début lorsqu'on utilise de nouveaux langages de programmation ou de nouveaux environnements de développement : la documentation en fait partie, et garder à l'esprit que l'on ne peut pas tout retenir.

ACCEDER A LA DOCUMENTATION D'UNITY

1. Dans l'onglet **HIERARCHY**, sélectionnez le ***GAMEOBJECT*** **Main Camera.**
2. Passez à l'onglet **INSPECTOR** et cliquez sur l'icône d'information (point d'interrogation, **?**) en haut à droite du composant **Transform :**

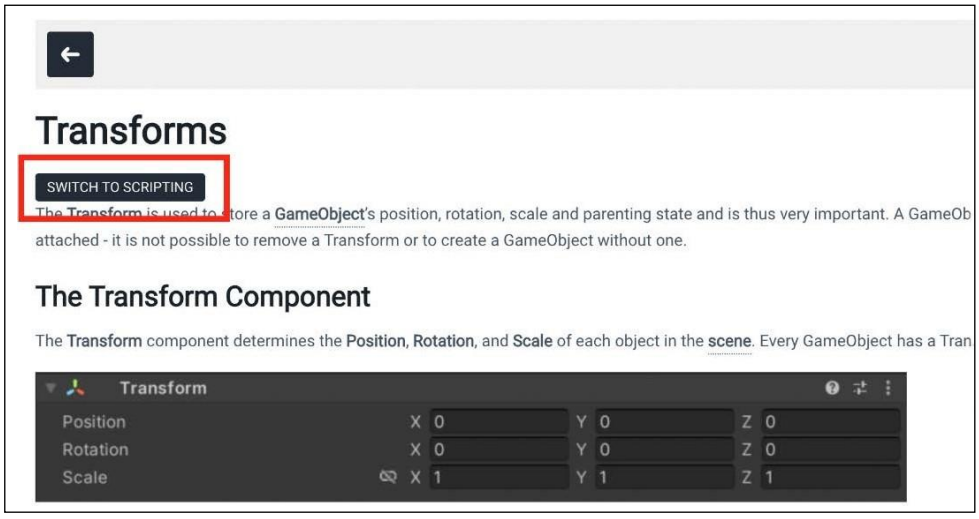


3. Vous verrez un navigateur web ouvert sur la page **Transforms** du manuel de référence :



Si nous voulons des exemples concrets de codage liés au composant **Transform** ? C'est très simple : il suffit de demander à la Référence de script :

1. Cliquez sur le lien **SWITCH TO SCRIPTING** sous le nom du composant ou de la classe (**Transforms**, dans ce cas) :



2. Ce faisant, le manuel de référence passe automatiquement à la référence des scripts :

3. Comme vous pouvez le constater, outre l'aide au codage, il existe également une option permettant de revenir au manuel de référence si nécessaire.

La Référence pour les scripts est un document volumineux, parce qu'il doit l'être. Cependant, cela ne signifie pas que vous devez le mémoriser ou même être familier avec toutes les informations qu'il contient pour commencer à écrire des scripts. Comme son nom l'indique, il s'agit d'une référence et non d'un test.

Si vous vous perdez dans la documentation, ou si vous ne savez pas où chercher, vous pouvez également trouver des solutions au sein de la riche communauté de développement Unity dans les endroits suivants :

- Forum Unity
- Unity Answers
- Unity Discord

LOCALISATION DES RESSOURCES C#

Documentation Microsoft Learn : contient un grand nombre de tutoriels, de guides de démarrage rapide et d'articles pratiques.

PROGRAMMATION

Tout langage de programmation est perçu au départ comme quelque chose de plutôt repoussant de prime abord, mais tous les langages de programmation sont constitués des mêmes éléments essentiels. Variables, méthodes et classes (ou objets) constituent l'ADN de la programmation conventionnelle ; la compréhension de ces concepts simples ouvre la voie à un monde entier d'applications diverses et complexes.

Si vous êtes novice en programmation, il y a beaucoup d'informations à assimiler au début, et il faudra ne pas hésiter à reprendre les notions, se les approprier en se fixant de objectifs très simples. Avec les nouvelles technologies, les exemples de codes sont nombreux, et s'il faut ne pas se priver de cette source, il faut rester humble et coder soi-même petit à petit : c'est le meilleur moyen d'apprendre de manière cyclique en copiant, en essayant, en recherchant ce qui ne va pas, etc …

DEFINITION DES VARIABLES

Commençons par une question simple : qu'est-ce qu'une variable ? Selon le point de vue que l'on adopte, il y a plusieurs façons de répondre à cette question :

- Sur le plan conceptuel, une variable est l'unité de base de la programmation, comme un atome l'est dans le monde physique (à l'exception de la théorie des cordes). Tout commence par des variables, et les programmes ne peuvent exister sans elles.
- Techniquement, une variable est une petite section de la mémoire de votre ordinateur qui contient une valeur **assignée**. Chaque variable garde une trace de l'endroit où ses informations sont stockées (c'est ce qu'on appelle une ADRESSE MEMOIRE), de sa valeur et de son type (par exemple, des nombres, des mots ou des listes).

- En pratique, une variable est un conteneur. Vous pouvez en créer de nouvelles quand vous le souhaitez, les remplir de choses, les déplacer, modifier ce qu'elles contiennent et y faire référence si nécessaire. Elles peuvent même être vides et rester utiles !

Complétez cette information par une recherche dans la documentation de Microsoft

Rédigez dans ce cadre

Les variables peuvent contenir différents types d'informations. En C#, les variables peuvent contenir des chaînes de caractères (texte), des entiers (nombres), des booléens (valeurs binaires représentant soit le vrai, soit le faux), et des objets comme ceux que l'on a vu jusqu'à présent (**GameObject** Main Camera et **component** transform)

Les noms sont importants

La mise en forme et le nom que l'on donne aux variables sont très importants car elles sont utilisées dans l'ensemble de l'application.

Les variables agissent comme des espaces réservés

Lorsque vous créez et nommez une variable, vous créez un espace réservé pour la valeur que vous souhaitez stocker.

UN PREMIER SCRIPT :

> Dans un projet nommé **PremierProjet**, la scène est nommée **PremiereScene**.
> Ajoutez un GameObject nommé **UnCube.**
> Ajoutez un script nommé **testA** : on notera que c'est un composant comme un autre.
> Liez le script **testA** au GameObject nommé **UnCube**

Pour ce faire, il faut :

> Visualiser dans l'arborescence de la fenêtre **PROJECT** votre script.
> Glisser le script et le déposer sur l'objet dans la fenêtre **HIERARCHY**.
> Vérifier dans **INSPECTOR** que le script apparaît bien comme composant.

Les directives using permettant à C# d'accéder à des bibliothèques UNITY.

La classe **testA** est basée sur une classe appelée *MonoBehaviour* qui va permettre à notre script de gérer l'objet et les composants qui lui sont attachés.

Start() et *Update()* sont des méthodes non obligatoires exécutées respectivement 1 fois au démarrage et 1 fois par Frame (Image) dont la fréquence est proportionnelle à la vitesse du processeur.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class testA : MonoBehaviour
{
    // Déclarations de variables au niveau de la classe
    // Start is called before the first frame update
    void Start()
    {
        // votre code ici
    }
    // Update is called once per frame
    void Update()
    {
        // votre code ici
    }
}
```

> Dans la méthode Start, ajoutez les lignes ci-dessous pour déclarer des variables

> ***ATTENTION : pensez dans MVS à sauvegarder systématiquement avant de retourner dans UNITY (CTRL + MAJ + S pour tout enregistrer), sinon, le script ne sera pas synchronisé avec Unity, et la compilation se fera avec l'ancienne version.***

```
public int Age = 30;
public int AnneeNaissance = 1965;
private int AnneeActuelle = 2023;
private string PhraseAffiche = string.Empty;
```

> Affichons la valeur des variables dans la console avec les lignes suivantes dans Start()

```
Debug.Log(Age);
Debug.Log("L'année de naissance est : " + AnneeNaissance);
```

> Exécutez le code en prenant soin d'activer la fenêtre **CONSOLE**.
> Observez les résultats qui doivent être conformes à ce que l'on attend du code.
> Stoppez le programme.
> Enregistrez vos fichiers dans MVS.

> ***Dans MVS, avant de basculer dans Unity, tapez SHIFT+CTRL+S***

Il est également important de noter que les variables dites *public* apparaissent dans l'inspecteur Unity, contrairement aux variables dites *private*.

Dans MVS, on peut voir que les variables utilisées dans le code sont en gras.

Quand vous cliquez sur Age, les occurrences de Age apparaissent en surbrillance.

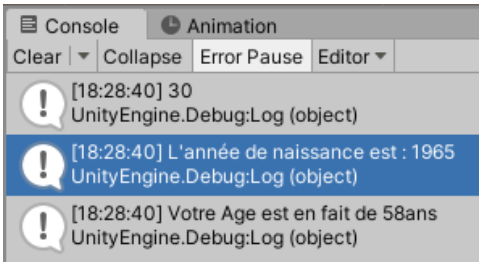
On peut remarquer que si on calcule l'âge, la variable Age ne devrait pas être de 30.

> Ajoutez le code dans Start() à la suite des lignes existantes.

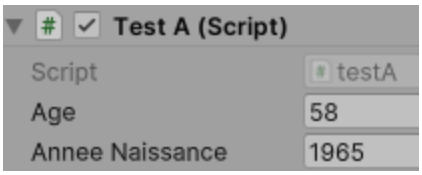
```
Age = AnneeActuelle - AnneeNaissance;
PhraseAffiche = "Votre Age est en fait de " + Age + " ans";
Debug.Log(PhraseAffiche);
```

> Exécutez le code

La console affiche bien la phrase attendue, la variable Age à été initialisée à 30, mais a été recalculée à 58.



De même, observez que dans l'inspecteur, les valeurs de nos variables s'affichent aussi : cela va permettre de modifier les valeurs de données et d'affficher des résultats de manière simple en alternative à Debug.Log(). La valeur 58 disparaît ensuite lorsque le programme est arrêté au profit de la valeur 30.



> Modifiez le code `public int Age = 30;` en `public int Age;`
> Exécutez à nouveau.
> Pendant l'exécution, modifiez la valeur de l'année de naissance.
> Stoppez l'exécution.

L'Age n'est pas recalculé, car la variable Age est affectée lors de l'exécution de Start(). L'instruction qui calcule l'âge n'est plus exécutée.

> Déplacez le calcul de l'Age dans Update()
> Testez en modifiant l'année de naissance.

L'inspecteur doit afficher l'Age.

> ***Pour commenter des lignes de codes, après les avoir sélectionnées, tapez CTRL+K+C***
> ***Pour retirer les commentaires, tapez CTRL+K+U***

> Mettez Debug.log en commentaire car la console est polluée par l'affichage infini du contenu de la variable PhraseAffiche.

COMPRENDRE LES METHODES

> ***Les méthodes sont à l'origine des actions***

Tout comme les variables, la définition des méthodes de programmation peut être fastidieuse ou dangereusement brève ; voici une autre approche en trois volets à envisager :

- D'un point de vue **conceptuel**, les méthodes sont la façon dont le travail est effectué dans une application.
- **Techniquement**, une méthode est un bloc de code contenant des instructions exécutables qui s'exécutent lorsque la méthode est appelée par son nom. Les méthodes peuvent recevoir des arguments (également appelés paramètres), qui peuvent être utilisés à l'intérieur de la portée de la méthode.
- En **pratique**, une méthode est un conteneur pour un ensemble d'instructions qui s'exécutent à chaque fois qu'elle est exécutée. Ces conteneurs peuvent également recevoir des variables en entrée, qui ne peuvent être référencées qu'à l'intérieur de la méthode elle-même.

Dans l'ensemble, les méthodes sont l'ossature de tout programme : elles relient tout et presque tout est construit à partir de leur structure.

LES METHODES EVITENT LA REPETITION DU CODE

En C#, une méthode est un bloc de code qui effectue une tâche spécifique lorsqu'elle est appelée. Elle peut prendre des paramètres en entrée et retourner une valeur en sortie, ou ne rien retourner du tout. Les méthodes sont généralement utilisées pour organiser le code en petites unités de fonctionnalité qui peuvent être appelées depuis d'autres parties du programme, ce qui permet de simplifier la logique du code et de le rendre plus facile à maintenir. Les méthodes peuvent également être utilisées pour encapsuler du code réutilisable qui peut être utilisé dans différentes parties du programme.

ATTENTION : vous allez ajouter un deuxième script : que faire du premier ? comment garder plusieurs scripts en cours d'exécution ?

Pour faire simple, le script associé à un objet peut être désactivé de sorte que si plusieurs scripts sont liés à un même objet, on peut facilement interrompre l'exécution de chaque script. On peut aussi créer un objet par script, mail faudra aussi désactiver l'objet (on peut le rendre invisible) ou le script. Au bout d'un certain temps, on peut aussi créer plusieurs scènes.

> Créez testB et liez testB au cube : par défaut un script est actif après avoir été lié.
> Désactivez testA du cube.
> Après avoir copié correctement le code dans testB, testez si tout fonctionne bien.

Observons le code ci-dessous de testB

```
public class testB : MonoBehaviour
{
    public int AgeActuel;
    public int AgeFutur;
    public int AnneeNaissance = 1965;
    private int AnneeActuelle = 2023;
    private int AnneeFutur = 2030;
    public string PhraseAffiche = string.Empty;

    void Update()
    {
        AgeActuel = CalculeAge(AnneeNaissance,AnneeActuelle);
        AgeFutur = CalculeAge(AnneeNaissance,AnneeFutur,ref PhraseAffiche);
    }
    private int CalculeAge(int dtNais, int dt)
    {
        int age = dt - dtNais;
        return age;
    }
    private int CalculeAge(int dtNais, int dt, ref string phrase)
    {
        int age = CalculeAge(dtNais, dt);
        phrase = "En " + dt + " vous avez " + age + " ans";
        return age;
    }
}
```

Nous avons une méthode CalculAge() qui calcule la différence entre deux années que l'on passe en paramètre ou argument, les paramètres passés à la méthode sont typés en entier (int ici).

La deuxième méthode qui porte le même nom est une **surcharge** de la première qui va permettre d'ajouter une **fonctionnalité** à la méthode de base : nous construisons une phrase qui va permettre un affichage en clair. Cette surcharge est un ajout d'une fonctionnalité optionnelle (mot clé **ref** pour permettre à la méthode d'accéder à **l'adresse mémoire** de la variable et en modifier sa valeur). Cela permet de ne pas dupliquer le code puisque la deuxième appelle la première et ajoute la nouvelle affectation de la variable phrase.

Le mot-clé (modificateur d'accès) private devant la méthode implique que celle-ci ne peut être appelée qu'à l'intérieur de la classe elle-même, pas à partir d'une autre classe.

PRESENTATION DES CLASSES

Les variables stockent des informations et les méthodes exécutent des actions, mais notre boîte à outils de programmation est encore quelque peu limitée. Nous avons besoin d'un moyen de créer une sorte de super conteneur, contenant des variables et des méthodes qui peuvent être référencées à l'intérieur du conteneur lui-même. C'est là qu'interviennent les classes :

- D'un point de vue conceptuel, une classe contient des informations, des actions et des comportements connexes dans un seul et même conteneur. Elles peuvent même communiquer entre elles.

- Techniquement, les classes sont des structures de données. Elles peuvent contenir des variables, des méthodes et d'autres informations programmatiques, qui peuvent toutes être référencées lorsqu'un objet de la classe est créé.
- En pratique, une classe est un modèle. Elle définit les règles et règlements applicables à tout objet (appelé instance) créé à l'aide du modèle de la classe.

UNE CLASSE UNITY COMMUNE

Chaque script créé dans Unity est une classe, comme le montre le mot-clé **class** :

```
public class testB : MonoBehaviour
{

}
```

MonoBehaviour signifie simplement que cette classe peut être attachée à un GameObject dans la scène Unity, et les deux crochets marquent les limites de la classe - tout le code à l'intérieur de ces crochets appartient à cette classe.

Les classes peuvent exister par elles-mêmes, ce que nous verrons lorsque nous créerons des classes autonomes

Les termes scripts et class sont parfois utilisés de manière interchangeable dans les ressources Unity.

> ***Par souci de cohérence les fichiers C# seront des scripts s'ils sont attachés à des GameObjects et des classes si ils sont autonomes.***

LES CLASSES SONT DES SCHEMAS DIRECTEURS

En C#, une classe est un modèle ou un plan pour créer des objets qui ont des propriétés et des comportements communs. Une classe est définie en regroupant des variables, des méthodes et d'autres membres de données et de fonctions qui sont utilisés pour créer des instances d'objets.

Les propriétés d'une classe sont les caractéristiques ou les attributs qui décrivent l'objet, comme son nom, son identifiant unique, sa couleur, sa taille, etc. Les méthodes sont les comportements ou les actions que l'objet peut effectuer, telles que l'affichage de ses propriétés, la modification de ses propriétés, la communication avec d'autres objets, etc.

EXEMPLE D'UNE CLASSE POUR GERER UN VEHICULE

Dans cet exemple, nous allons créer une classe "Vehicle" (véhicule) avec des propriétés telles que la vitesse, la direction, la couleur et une méthode pour changer la direction.

```
public class Vehicle
{
    // Propriétés
    public double Speed { get; set; }
    public string Direction { get; set; }
    public string Color { get; set; }

    // Méthodes
    public void ChangeDirection(string newDirection)
    {
        this.Direction = newDirection;
    }
}
```

Dans cette classe, nous avons trois propriétés: `Speed`, `Direction`, et `Color`. `Speed` est une propriété numérique qui stocke la vitesse actuelle du véhicule. `Direction` est une propriété de chaîne de caractères qui stocke la direction actuelle du véhicule. `Color` est une propriété de chaîne de caractères qui stocke la couleur du véhicule.

La méthode `ChangeDirection` permet de changer la direction du véhicule en prenant une chaîne de caractères en entrée qui représente la nouvelle direction.

Vous pouvez utiliser cette classe pour créer des instances de véhicules spécifiques, en définissant leurs propriétés et en appelant leurs méthodes.

Voici un exemple de code qui crée une instance de la classe `Vehicle` et utilise sa méthode `ChangeDirection` pour changer sa direction :

```
Vehicle myVehicle = new Vehicle();
myVehicle.Speed = 50.0;
myVehicle.Direction = "north";
myVehicle.Color = "red";

myVehicle.ChangeDirection("east");
```

TRAVAILLER AVEC DES COMMENTAIRES

Les commentaires sont essentiels pour la lisibilité, la compréhension, la maintenance et le débogage du code. En ajoutant des commentaires clairs et concis, vous pouvez aider les autres développeurs à comprendre votre code plus rapidement et efficacement, tout en améliorant la qualité globale de votre code.

```
// permet de mettre une ligne en commentaire
/* ceci est un
commentaire sur
plusieurs lignes */
```

En C#, il est recommandé de documenter chaque méthode à l'aide de commentaires XML pour améliorer la lisibilité, la compréhension et la maintenance du code. Les commentaires XML sont des commentaires spéciaux qui commencent par trois barres obliques (///), suivis d'une description de la méthode et de ses paramètres, ainsi que des informations sur les exceptions potentielles qui peuvent être levées.

Voici un exemple de documentation d'une méthode C# à l'aide de commentaires XML.

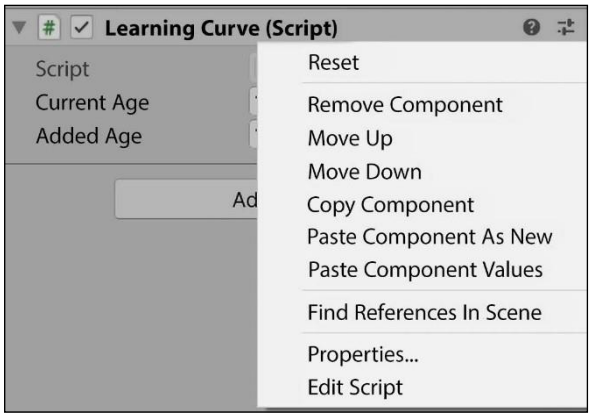
Une fois mis en place, les informations fournies seront affichées lors du passage de la souris sur les appels de la méthode et ses paramètres.

```
    /// <summary>
    /// Calcule la différence d'age entre une année dt et une année de naissance dtNais
    /// </summary>
    /// <param name="dtNais">Année de naissance</param>
    /// <param name="dt">Année de référence</param>
    /// <returns>Age calculé</returns>
    private int CalculeAge(int dtNais, int dt)
    {
        int age = dt - dtNais;
        return age;
    }
```

Lorsque vous êtes en **mode développement**, toutes les modifications que vous apportez aux variables sont sauvegardées par Unity. Cela signifie que si vous quittez Unity et que vous le redémarrez, les modifications seront conservées.

Les modifications que vous apportez aux valeurs du panneau **Inspecteur** en **mode lecture** ne modifient pas votre script, mais elles remplacent les valeurs que vous avez attribuées à votre script en **mode développement**.

Toutes les modifications apportées en **mode lecture** seront toujours réinitialisées automatiquement lorsque vous arrêterez le **mode lecture**. Si vous devez annuler des modifications effectuées dans le panneau **Inspecteur**, vous pouvez rétablir les valeurs par défaut (parfois appelées valeurs **initiales**) du script. Cliquez sur l'icône à trois points verticaux située à droite de n'importe quel composant, puis sélectionnez **Réinitialiser**, comme indiqué dans la capture d'écran suivante :



Cela devrait vous donner une certaine tranquillité d'esprit : si vos variables deviennent incontrôlables, il y a toujours la possibilité de réinitialiser le système.

UN COUP DE POUCE DE MONOBEHAVIOUR

Tous les scripts n'ont pas besoin d'hériter de MonoBehaviour - ce n'est nécessaire que pour ceux que vous voulez ajouter aux GameObjects dans vos scènes Unity.

Le sujet de l'héritage de classes est un peu avancé à ce stade de votre parcours de programmation ; considérez-le comme la classe MonoBehaviour qui prête quelques-unes de ses variables et méthodes à votre script.

Nous verrons plus loin comment écrire des classes qui n'héritent pas de MonoBehaviour.

Cherchez des informations sur Start() et Update() dans la documentation Unity. Existe-t-il d'autres méthodes spécifiques à Unity.

Votre réponse ici.

Cherchez la documentation sur MonoBehaviour

Votre réponse ici.

DEBOGAGE DU CODE

Nous avons travaillé avec `Debug.log()` pour afficher des données ou des variables dans la console. Cela fonctionne aussi pour les objets.

Pour un débogage plus complexe, utilisez `Debug.LogFormat()` : cela vous permettra de placer des variables à l'intérieur du texte imprimé en utilisant des espaces réservés.

V1 : Ceux-ci sont marqués par une paire de crochets, chacun contenant chacun le nom de la variable.

V2 : Une alternative consiste à placer entre crochet un n° d'ordre, puis de placer les variables en paramètres à passer dans l'ordre fixé.

> Dans le script testB, ajoutez ou modifiez la procédure Start()

```
private void Start()
{
    // V1
    Debug.LogFormat($"Naissance : {AnneeNaissance} - Année Actuelle : {AnneeActuelle}");
    // V2
    Debug.LogFormat("Naissance : {0} - Année Actuelle : {1} », AnneeNaissance, AnneeActuelle);
}
```

COMPRENDRE LES VARIABLES

Dans le chapitre précédent, nous avons vu comment les variables sont écrites et nous avons abordé les fonctionnalités de haut niveau qu'elles fournissent. Cependant, il nous manque encore la syntaxe qui rend tout cela possible, alors commençons par les bases : la déclaration des variables.

DECLARER DES VARIABLES

Les variables n'apparaissent pas simplement au début d'un script C# ; elles doivent être déclarées conformément à certaines règles et exigences. À son niveau le plus élémentaire, une déclaration de variable doit satisfaire aux exigences suivantes :

- Le type de données que la variable stockera doit être spécifié
- La variable doit avoir un nom unique
- S'il existe une valeur assignée, elle doit correspondre au type spécifié
- La déclaration de la variable doit se terminer par un point-virgule

Le respect de ces règles se traduit par la syntaxe suivante : `dataType UniqueName = valeur ;`

Les variables doivent porter des noms uniques afin d'éviter tout conflit avec des mots qui ont déjà été pris par C#, et qui sont appelés **mots-clés**. Vous trouverez la liste complète des mots-clés protégés dans la documentation Microsoft.

DECLARATIONS DE TYPES ET DE VALEURS

Le scénario le plus courant pour la création de variables est celui dans lequel toutes les informations requises sont disponibles au moment de la déclaration. Par exemple, si nous connaissons l'âge d'un joueur, l'enregistrer serait aussi simple que de faire ce qui suit : `int AgeActuel = 32 ;`

Dans ce cas, toutes les conditions de base sont remplies :

- Un type de données est spécifié, à savoir **int** (abréviation de **integer**, qui est un mot pour désigner un nombre entier).
- Un nom unique est utilisé, à savoir `AgeActuel`
- 32 est un nombre entier, qui correspond au type de données spécifié
- La déclaration se termine par un point-virgule

Cependant, dans certains cas, vous voudrez déclarer une variable sans connaître immédiatement sa valeur. Nous aborderons ce sujet dans la section suivante.

DECLARATIONS DE TYPE UNIQUEMENT

Considérez un autre scénario : vous connaissez le type de données que vous souhaitez stocker dans une variable ainsi que son nom, mais pas sa valeur. La valeur sera calculée et affectée ailleurs, mais vous devez toujours déclarer la variable en tête du script. Cette situation est : `int AgeActuel;`

Seuls le type (int) et le nom unique (`AgeActuel`) sont définis, mais l'instruction reste valide car nous avons respecté les règles. Si aucune valeur n'est attribuée, des valeurs par défaut seront attribuées en fonction du type de la variable. Dans ce cas, `AgeActuel` sera mis à 0, ce qui correspond au type **int**. Dès que la valeur réelle de la variable devient disponible, elle peut facilement être définie dans une instruction séparée en faisant référence au nom de la variable et en lui attribuant une valeur : `AgeActuel = 32 ;`

UTILISATION DES MODIFICATEURS D'ACCES

En C#, les modificateurs d'accès sont utilisés pour contrôler la visibilité des membres d'une classe, c'est-à-dire leur accessibilité à partir d'autres parties du code. Il existe quatre modificateurs d'accès en C# : public, private, protected et internal.

- **public** : Les membres déclarés avec le modificateur public sont accessibles à partir de n'importe quelle partie du programme. Les membres publics peuvent être appelés à partir de n'importe quelle classe ou méthode, qu'elles soient situées dans le même projet ou dans un projet différent.
- **private** : Les membres déclarés avec le modificateur private sont accessibles uniquement à partir de l'intérieur de la classe dans laquelle ils sont déclarés. Les membres privés ne peuvent pas être appelés depuis une autre classe ou méthode, même si elles sont situées dans le même projet.
- **protected** : Les membres déclarés avec le modificateur protected sont accessibles à partir de la classe dans laquelle ils sont déclarés et de ses classes dérivées (héritées). Les membres protégés ne peuvent pas être appelés à partir d'une classe qui n'est pas dérivée de la classe dans laquelle ils sont déclarés.
- **internal** : Les membres déclarés avec le modificateur internal sont accessibles à partir de n'importe quelle partie du code dans le même assemblage (un assemblage est un fichier .dll ou .exe). Les membres internes ne peuvent pas être appelés depuis un autre assemblage.

Il est important de noter que la visibilité par défaut des membres en C# est private. Cela signifie que si vous ne spécifiez pas de modificateur d'accès pour un membre, il sera automatiquement déclaré comme privé. Il est donc recommandé d'utiliser explicitement le modificateur d'accès approprié pour chaque membre de votre classe afin de rendre votre code plus clair et plus facilement compréhensible.

Toute variable qui n'est pas marquée comme **public** est considérée comme **private** par défaut et n'apparaîtra pas dans la base de données Unity dans la fenêtre **INSPECTOR**.

> ***Si vous incluez un modificateur, la recette syntaxique mise à jour se présentera comme suit*** `: accessModifier dataType UniqueName = value ;`

TRAVAILLER AVEC DES TYPES

L'attribution d'un type spécifique à une variable est un choix important, qui se répercute sur toutes les interactions de la variable au cours de sa vie. Le C# étant ce qu'on appelle un langage fortement typé, chaque variable doit avoir un type de données sans exception. En comparaison, les langages de programmation comme JavaScript, par exemple, ne sont pas sûrs quant au type de données. Cela signifie qu'il existe des règles spécifiques lorsqu'il s'agit d'effectuer des opérations avec certains types, et des règles lorsqu'il s'agit de convertir un type de variable donné en un autre.

TYPES D'ELEMENTS INTEGRES COURANTS

Tous les types de données dans C# descendent (ou **dérivent**, en termes programmatiques) d'un ancêtre commun : **SYSTEM.OBJECT**. Cette hiérarchie, appelée Common Type System (CTS), signifie que les différents types ont de nombreuses fonctionnalités communes. Le tableau suivant présente certaines des options de type de données les plus courantes et les valeurs qu'elles stockent :

Type	Contents of the variable
int	A simple integer, such as the number 3
float	A number with a decimal, such as the number 3.14
string	Characters in double quotes, such as,"Watch me go now"
bool	A Boolean, either **true** or **false**

En plus de spécifier le type de valeur qu'une variable peut stocker, les types contiennent des informations supplémentaires sur eux-mêmes, notamment les suivantes :

- Espace de stockage requis
- Valeurs minimales et maximales
- Opérations autorisées
- Emplacement dans la mémoire

- Méthodes accessibles
- Type de base (dérivé)

Travailler avec tous les types offerts par C# est un exemple parfait d'utilisation de la documentation plutôt que de la mémorisation. Très vite, l'utilisation des types personnalisés, même les plus complexes, vous semblera une seconde nature.

CONVERSIONS DE TYPES

Nous avons déjà vu que les variables ne peuvent contenir que des valeurs de leur type déclaré, mais il peut arriver que vous deviez combiner des variables de types différents. Dans la terminologie de la programmation, on appelle cela des conversions, qui se présentent sous deux formes principales :

Les conversions **implicites** ont lieu automatiquement, généralement lorsqu'une valeur plus petite peut être insérée dans un autre type de variable sans qu'il soit nécessaire de l'arrondir. Par exemple, tout nombre entier peut être implicitement converti en valeur double ou flottante sans code supplémentaire :

```
int MyInteger = 3;
float MyFloat = MyInteger;
Debug.Log(MyInteger);
Debug.Log(MyFloat);
```

Les conversions **explicites** sont nécessaires lorsqu'il y a un risque de perdre les informations d'une variable lors de la conversion. Par exemple, si nous voulions convertir une valeur double en valeur int, nous devrions la convertir explicitement en ajoutant le type de destination entre parenthèses avant la valeur que nous voulons convertir.

Cela indique au compilateur que nous sommes conscients que des données (ou de la précision) peuvent être perdues :

```
int ExplicitConversion = (int)3.14 ;
```

Dans cette conversion explicite, 3.14 serait arrondi à 3, perdant ainsi les valeurs décimales.

C# fournit des méthodes intégrées pour convertir explicitement des valeurs en types courants. Par exemple, n'importe quel type peut être converti en chaîne de caractères avec la méthode ToString(), tandis que la classe Convert peut gérer des conversions plus complexes.

DECLARATIONS DEDUITES

Heureusement, C# peut déterminer le type d'une variable à partir de la valeur qui lui est attribuée. Par exemple, le mot-clé var peut indiquer au programme que le type de la donnée, Temperature, doit être déterminé par sa valeur de 32.0, qui est un nombre décimal : `var Temperature = 32.0f ;`

Bien que cela soit pratique dans certaines situations, ne vous laissez pas entraîner dans une habitude de programmation paresseuse consistant à utiliser des déclarations de variables déduites pour tout. Cela ajoute beaucoup d'incertitude à votre code, alors qu'il devrait être clair comme de l'eau de roche. Les déclarations de variables déduites ne doivent être utilisées que lorsque vous testez un code et que vous ne connaissez pas le type de données stockées. Une fois que vous le savez, il est recommandé de modifier la déclaration de la variable en fonction du type spécifique afin d'éviter les erreurs d'exécution ultérieures.

TYPES PERSONNALISES

Lorsque nous parlons de types de données, il est important de comprendre dès le départ que les nombres et les mots (appelés valeurs littérales) ne sont pas les seuls types de valeurs qu'une variable peut stocker. Par exemple, une classe, une structure ou une énumération peuvent être stockées en tant que variables.

NOMMER LES VARIABLES (CASSE CAMEL)

La casse Camel (ou notation en chameau en français) est une convention de nommage pour les identificateurs de variables, de méthodes, de classes et d'autres éléments de code en informatique. La casse Camel est largement utilisée en C# et dans d'autres langages de programmation tels que Java et JavaScript.

La casse Camel consiste à écrire le premier mot en minuscules, puis chaque mot suivant en majuscules sans espace, de sorte que le nom ressemble à une bosse de chameau. Par exemple, voici quelques exemples d'identificateurs de code en casse Camel :

- firstName
- lastName
- accountBalance
- orderDetails
- customerAddress

La casse Camel est utilisée pour améliorer la lisibilité du code en rendant les identificateurs plus faciles à lire. En utilisant des majuscules pour séparer les mots, les identificateurs sont plus faciles à distinguer les uns des autres et à identifier leurs composants individuels.

Il est important de noter que la casse Camel est une convention de nommage et n'affecte pas le comportement du code lui-même. Cependant, il est recommandé de suivre les conventions de nommage telles que la casse Camel pour améliorer la lisibilité et la compréhension du code par les autres développeurs qui travaillent sur le même projet.

NOMMER LES VARIABLES (CASSE PASCAL)

La casse Pascal consiste à écrire le premier mot et chaque mot suivant en majuscules sans espace. Contrairement à la casse Camel, qui met le premier mot en minuscules, la casse Pascal met tous les mots en majuscules. Par exemple, voici quelques exemples d'identificateurs de code en casse Pascal :

- FirstName
- LastName
- AccountBalance
- OrderDetails
- CustomerAddress

COMPRENDRE LA PORTEE DES VARIABLES

À l'instar des modificateurs d'accès, qui déterminent les classes extérieures pouvant accéder aux informations d'une variable, la portée d'une variable est le terme utilisé pour décrire l'emplacement d'une variable donnée et son point d'accès au sein de la classe qui la contient.

Il existe trois niveaux principaux de portée des variables en C :

- La portée globale se réfère à une variable qui peut être accédée par un programme entier. C# ne supporte pas directement les variables globales, mais le concept est utile dans certains cas
- La portée d'une classe ou d'un membre fait référence à une variable qui est accessible partout dans la classe qui la contient.
- La portée locale fait référence à une variable qui n'est accessible qu'à l'intérieur d'une méthode ou d'un bloc spécifique du code dans lequel il est créé.

PRESENTATION DES OPERATEURS

Dans les langages de programmation, les symboles d'opérateurs représentent les fonctions arithmétiques, d'affectation, relationnelles et logiques que les types peuvent exécuter. Les opérateurs arithmétiques représentent les fonctions mathématiques de base, tandis que les opérateurs d'affectation exécutent à la fois des fonctions mathématiques et d'affectation sur une valeur donnée. Les opérateurs relationnels et logiques évaluent les conditions entre plusieurs valeurs, comme plus grand que, moins que et égal à.

ARITHMETIQUE ET DEVOIRS

Les symboles des opérateurs arithmétiques vous sont déjà familiers : + pour l'addition,- pour la soustraction,/ pour la division,* pour la multiplication

Les opérateurs de C# suivent l'ordre conventionnel des opérations, c'est-à-dire qu'ils évaluent d'abord les parenthèses, puis les exposants, puis la multiplication, puis la division, puis l'addition et enfin la soustraction.

Par exemple, les équations suivantes donneront des résultats différents, même si elles contiennent les mêmes valeurs et les mêmes opérateurs :

5 + 4 - 3 / 2 * 1 = 8

5 + (4 - 3) / 2 * 1 = 5

Les opérateurs fonctionnent de la même manière lorsqu'ils sont appliqués à des variables qu'à des valeurs littérales. Les opérateurs d'affectation peuvent être utilisés en remplacement de n'importe quelle opération mathématique en utilisant n'importe quel symbole arithmétique et égal ensemble. Par exemple, si nous voulions multiplier une variable, vous pourriez utiliser le code suivant :

```
int x = 32;
x = x * 2;
```

La deuxième façon de procéder, alternative, est présentée ici :

```
int x = 32 ;
x *= 2 ;
```

Le symbole égal est également considéré comme un opérateur d'affectation dans C#. Les autres symboles d'affectation suivent le même schéma syntaxique que notre exemple de multiplication précédent : +=, -=, et /= pour ajouter et assigner, soustraire et assigner, et diviser et assigner, respectivement.

Les chaînes de caractères sont un cas particulier en ce qui concerne les opérateurs, car elles peuvent utiliser le symbole d'addition pour créer un texte en patchwork, comme suit : `string NomComplet = "Toto " + "Dupont" ;`

Notez que les opérateurs arithmétiques ne fonctionnent pas avec tous les types de données. Par exemple, les opérateurs * et / ne fonctionnent pas sur les chaînes de caractères, et aucun de ces opérateurs ne fonctionne sur les booléens. Ayant appris que les types ont des règles qui régissent le type d'opérations et d'interactions qu'ils peuvent avoir, nous allons tenter notre

DEFINITION DES METHODES

Dans le chapitre précédent, nous avons brièvement abordé le rôle que jouent les méthodes dans nos programmes, à savoir qu'elles stockent et exécutent des instructions, tout comme les variables stockent des valeurs. Nous devons maintenant comprendre la syntaxe des déclarations de méthodes et la manière dont elles conduisent l'action et le comportement dans nos classes.

Comme pour les variables, les déclarations de méthodes ont leurs exigences de base, qui sont les suivantes :

- Le type de données qui seront renvoyées par la méthode (les méthodes ne doivent pas toutes renvoyer quelque chose, donc cela peut être void).
- Un nom unique, commençant par une majuscule
- Une paire de parenthèses après le nom de la méthode
- Une paire de parenthèses marquant le corps de la méthode (où les instructions sont stockées)

DECLARATION DES METHODES

Les méthodes peuvent également disposer des quatre mêmes modificateurs d'accès que les variables, ainsi que des paramètres d'entrée. Les paramètres sont des variables qui peuvent être transmises aux méthodes et auxquelles on peut accéder à l'intérieur de celles-ci. Le nombre de paramètres d'entrée que vous pouvez utiliser n'est pas limité, mais chacun doit être séparé par une virgule, indiquer son type de données et avoir un nom unique.

Exemple d'une méthode (équivaut à une fonction) qui retourne un résultat dans la variable **result** :

```
accessModifier returnType UniqueName(parameterType parameterName)
{
   // Corps de la méthode
   returnType result ;
   // inst 1
   // inst 2
   ……
   return result ;
}
```

Exemple d'une méthode qui ne retourne rien (équivaut à une procédure) :

```
accessModifier void UniqueName(parameterType parameterName)
{
   // Corps de la méthode

   // inst 1
   // inst 2
   ……
}
```

CONVENTIONS D'APPELLATION

Comme les variables, les méthodes ont besoin de noms uniques et significatifs pour les distinguer dans le code. Les méthodes conduisent à des actions, c'est donc une bonne pratique de les nommer en gardant cela à l'esprit. Par exemple, FonctionRepartition() ou DensiteProba() ressemble à une commande, ce qui se lit bien lorsque vous l'appelez dans un script, alors qu'un nom tel que Resume() ou FaitLe() est insipide et ne donne pas une image très claire de ce que la méthode va accomplir. Comme les variables, les noms de méthodes sont écrits dans la casse Pascal.

EXECUTION SEQUENTIELLE DU CODE

Nous avons vu que les lignes de code s'exécutent séquentiellement dans l'ordre où elles sont écrites L'appel d'une méthode demande au programme de faire un détour par les instructions de la méthode, de les exécuter une à une, puis de reprendre l'exécution séquentielle à l'endroit où la méthode a été appelée.

SPECIFICATION DES PARAMETRES

Dans les méthodes CalculAge(), nous avons utilisé des paramètres.

> Dans un script nommé testC, réfléchissez à une classe Personne : quelles membres (propriétés et méthodes) pourriez-vous imaginer. Pour les propriétés, pensez à ce qui est nécessaire pour la décrire en tant qu'être humain, personnage de jeu, etc … Pour les méthodes, que fait la personne comme action, est-ce un étudiant ? un enseignant ? un joueur ? un personnage non joueur ? (amical, ennemi) : de quelles paramètres la méthode a-t-elle besoin pour exécuter l'action que vous avez défini.

EXEMPLE (SOURCE CHATGPT)

```
using UnityEngine;
public class Player : MonoBehaviour
{
    // Champs privés
    private string playerName;
    private int playerLevel;
    private int playerHealth;

    // Propriétés publiques en lecture seule
    public string PlayerName => playerName;
    public int PlayerLevel => playerLevel;
    public int PlayerHealth => playerHealth;

    // Méthode publique pour prendre des dégâts
    public void TakeDamage(int damage)
    {
        playerHealth -= damage;
        if (playerHealth <= 0)
        {
            Die();
        }
    }

    // Méthode publique pour tuer le joueur
    public void Die()
    {
        Destroy(gameObject);
    }

    // Méthode publique pour augmenter le niveau du joueur
    public void LevelUp()
    {
        playerLevel++;
        playerHealth = Mathf.RoundToInt(playerHealth * 1.5f);
    }

    // Méthode publique pour initialiser le joueur avec un nom, un niveau et une santé donnés
    public void InitializePlayer(string name, int level, int health)
    {
        playerName = name;
        playerLevel = level;
        playerHealth = health;
    }
}
```

Dans cet exemple, nous avons déclaré une classe Player qui hérite de la classe MonoBehaviour du moteur Unity. La classe contient des champs privés pour stocker le nom, le niveau et la santé du joueur, ainsi que des propriétés publiques en lecture seule pour permettre l'accès aux champs sans permettre la modification directe.

Nous avons également défini plusieurs méthodes publiques pour effectuer des actions sur le joueur, telles que prendre des dégâts, mourir, augmenter de niveau et initialiser le joueur avec des valeurs spécifiques. Les méthodes sont toutes publiques afin qu'elles puissent être appelées depuis d'autres classes.

En utilisant cette classe, nous pouvons créer un objet joueur dans le moteur Unity et l'initialiser en appelant la méthode InitializePlayer avec des valeurs appropriées. Ensuite, nous pouvons appeler d'autres méthodes pour effectuer des actions sur le joueur en réponse à des événements dans le jeu, tels que prendre des dégâts, augmenter de niveau ou mourir.

En résumé, une classe pour un joueur en utilisant le moteur Unity peut contenir des propriétés, des champs et des méthodes pour stocker et manipuler les données du joueur. Les méthodes peuvent être utilisées pour effectuer des actions sur le joueur en réponse à des événements dans le jeu, tels que prendre des dégâts, augmenter de niveau ou mourir.

SPECIFICATION DES VALEURS DE RETOUR

En plus d'accepter des paramètres, les méthodes peuvent renvoyer des valeurs de n'importe quel type C#.

Les méthodes dont le type de retour est void peuvent toujours utiliser le mot-clé return sans qu'aucune valeur ou expression ne leur soit attribuée. Lorsque la ligne contenant le mot-clé return est atteinte, la méthode cesse d'être exécutée. Ceci est utile dans les cas où vous souhaitez vérifier l'existence d'une ou plusieurs valeurs avant de continuer ou vous prémunir contre les plantages de programme.

UTILISATION DES VALEURS DE RETOUR

Lorsqu'il s'agit d'utiliser les valeurs de retour, deux approches sont possibles :

- Créer une variable locale pour capturer (stocker) la valeur retournée.
- Utilisez la méthode d'appel elle-même comme substitut de la valeur renvoyée, en l'utilisant comme une variable. La méthode d'appel est la ligne de code qui déclenche les instructions, qui, dans notre exemple, serait GenerateCharacter("Spike", CharacterLevel). Vous pouvez même passer une méthode d'appel à une autre méthode en tant qu'argument si nécessaire.

La première option est préférée dans la plupart des cercles de programmation pour sa lisibilité. Le fait de jeter des appels de méthode sous forme de variables peut rapidement devenir problématique, surtout lorsque nous les utilisons comme arguments dans d'autres méthodes.

EXEMPLE : CONVERSION CELSIUS EN FAHRENHEIT

> Ajoutez un script testConvDegCelsius.
> Ajoutez une méthode qui retourne les degrés Fahrenheit avec les degrés Celsius en paramètre.

La méthode :

```
public float ConvertCelsiusToFahrenheit(float degC)
{
    float degF;
    degF = (degC * 1.8f) + 32f;
    return degF;
}
```

La méthode pourra être appelée en dehors de la classe, on lui passera le paramètre de type float degC, on utilise une variable locale degF de type float pour faire le calcul. La méthode retournera cette variable locale.

Appeler la méthode :

```
void Start()
{
    float DegC = 30.0f;
    // Pour faire un test simple, appeler la méthode et vérifier ce qui se passe, on peut simplement faire :
    Debug.Log(ConvertCelsiusToFahrenheit(DegC));
   // déclarer une variable locale pour recueillir le résultat, puis afficher la variable DegF
   float DegF = ConvertCelsiusToFahrenheit(DegC);
  Debug.Log(DegF) ;
}
```

> Modifiez le code pour afficher degC et degF dans l'inspecteur
> Test n°1 : La conversion sera faite seulement au démarrage de la scène.
> Test n°2 : La conversion sera faite en continu.

COMPRENDRE LES METHODES COURANTES D'UNITY

Nous avons vu les méthodes par défaut les plus courantes qui sont livrées avec un script Unity C# : Start() et Update(). Contrairement aux méthodes que nous définissons nous-mêmes, les méthodes appartenant à la classe MonoBehaviour sont appelées

automatiquement par le moteur Unity selon leurs règles respectives. Dans la plupart des cas, il est important d'avoir au moins une méthode MonoBehaviour dans un script pour lancer votre code.

Vous avez fait à la fin du chapitre 1 une recherche dans la documentation Unity : on y trouve une liste assez impressionnante de méthodes disponibles. Il est important de regarder l'ordre dans lequel les méthodes sont exécutées par UNITY.

Lorsque l'on veut apprendre la programmation de base en C#, Unity nous offre un environnement de développement très simple et puissant et nous n'avons pas besoin d'approfondir plus que nécessaire ce qui est disponible via MonoBehaviour.

Lorsque nous voulons aller plus loin en programmation, profiter de MonoBehaviour avec la compréhension des objets et composants est vraiment intéressant. Unity est un véritable outil de développement performant, accessible gratuitement, énormément documenté et utilisable avec d'autres langages ou outils : appeler des scripts python, aller chercher sur un serveur Raspberry des données de capteur via MQTT (ou autre protocole réseau), construire une interface utilisateur. On peut le voir comme un fédérateur ou une alternative à d'autres outils moins bien suivis, moins fiables, plus difficiles à mettre en œuvre. On peut grâce à des scripts communiquer directement avec Arduino, même s'il sera plus pertinent de coder l'acquisition de données en langage Arduino et rendre disponible les données avec Unity. De manière professionnelle, les applications industrielles sont bien sûr très développées dans tous les domaines (Réalité Virtuelle ou augmentée, modes opératoires, simulation, etc).

Il faut progresser lentement et mettre en œuvre des choses simples : nous allons essayer de séparer aussi ce que l'on apprend au niveau de la programmation ou des mécanismes du moteur Unity.

LA METHODE START()

Unity appelle la méthode Start() sur la première image où un script est activé pour la première fois. Comme les scripts MonoBehaviour sont presque toujours attachés à des **GAMEOBJECTS** dans une scène, leurs scripts attachés sont activés en même temps qu'ils sont chargés lorsque vous appuyez sur **Play**. Dans notre projet, vous avez associé les scripts à un ou plusieurs objets : ils sont activés ou non afin de ne pas surcharger notre fenêtre **CONSOLE**.

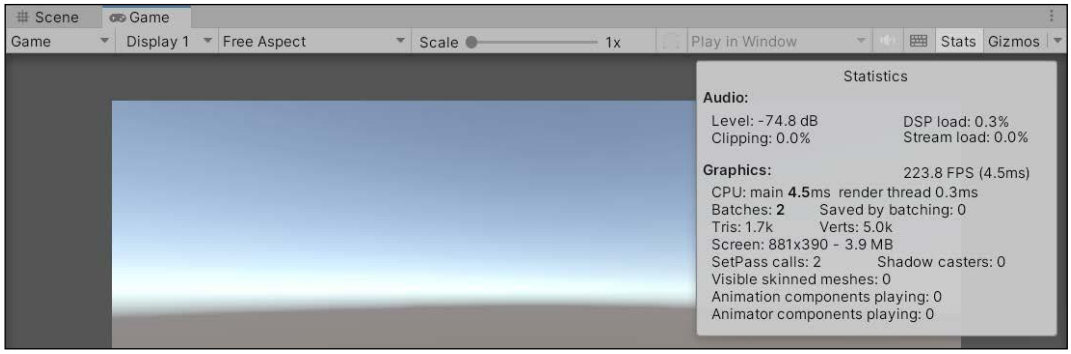
Start ne se déclenche qu'une seule fois, ce qui en fait un excellent outil pour afficher des informations ponctuelles sur la console et faire des essais de code.

LA METHODE UPDATE()

Lors de l'exécution du jeu, la fenêtre de la **scène** est affichée plusieurs fois par seconde, ce que l'on appelle le **taux de rafraîchissement** ou le nombre **d'images par seconde** (**FPS)**.

Après l'affichage de chaque image, la méthode Update() est appelée par Unity, ce qui en fait l'une des méthodes les plus exécutées dans votre jeu. Elle est donc idéale pour détecter les entrées de souris et de clavier ou pour exécuter la logique de jeu.

Pour connaître le nombre de FPS sur votre machine, lancez la lecture dans Unity et cliquez sur l'onglet **Stats** dans le coin supérieur droit de l'affichage du **jeu :**



Exercice 50-1

FLUX DE CONTROLE ET COLLECTIONS

L'une des principales fonctions d'un ordinateur est de contrôler ce qui se passe lorsque des conditions prédéterminées sont remplies. Lorsque vous cliquez sur un dossier, vous vous attendez à ce qu'il s'ouvre ; lorsque vous tapez au clavier, vous vous attendez à ce que le texte reflète votre frappe. L'écriture du code C# pour les applications ou les jeux n'est pas différent : ils doivent tous deux se comporter d'une certaine manière dans un état donné, et d'une autre manière lorsque les conditions changent. En termes de programmation, c'est ce qu'on appelle le flux de contrôle, ce qui est approprié car il contrôle le flux d'exécution du code dans différents scénarios.

En plus de travailler avec des instructions de contrôle, nous allons nous pencher sur les types de données de collection. Les collections sont une catégorie de types qui permettent de stocker plusieurs valeurs et groupes de valeurs dans une seule variable. Ce chapitre est divisé en plusieurs parties :

- Tests conditionnels
- Travailler avec des collections de tableaux, de dictionnaires et de listes
- Instructions d'itération avec les boucles for, foreach et while
- Correction des boucles infinies

TESTS CONDITIONNELS

Les problèmes de programmation les plus complexes peuvent souvent se résumer à un ensemble de choix simples qu'un jeu ou un programme évalue et sur lesquels il agit. Visual Studio et Unity n'étant pas en mesure de faire ces choix par eux-mêmes, c'est à vous qu'il revient d'écrire ces décisions.

Les instructions de sélection if-else et switch vous permettent de spécifier des chemins de dérivation, basés sur une ou plusieurs conditions, et les actions que vous souhaitez entreprendre dans chaque cas. Traditionnellement, ces conditions sont les suivantes

- Détection de l'entrée de l'utilisateur
- Évaluation d'expressions et logique booléenne
- Comparaison de variables ou de valeurs littérales

Dans la section suivante, vous commencerez par la plus simple de ces instructions conditionnelles, if-else.

L'INSTRUCTION IF-ELSE

UN EXEMPLE SIMPLE D'UTILISATION DE LA STRUCTURE DE CONTROLE "IF" EN C#

```
int age = 18;
if (age >= 18)
{
    Debug.Log("Vous êtes majeur !");
}
else
{
    Debug.Log("Vous êtes mineur !");
}
```

Dans cet exemple, nous avons une variable age initialisée à 18. Nous utilisons ensuite la structure de contrôle "if" pour vérifier si age est supérieur ou égal à 18. Si c'est le cas, nous affichons le message "Vous êtes majeur !" à l'aide de la méthode Console.WriteLine(). Sinon, nous affichons le message "Vous êtes mineur !".

Le code entre les accolades après le mot clé "if" est exécuté si la condition entre parenthèses est vraie. Le code entre les accolades après le mot clé "else" est exécuté si la condition est fausse.

Il est important de noter que la condition entre parenthèses dans la structure "if" doit être une expression booléenne, c'est-à-dire une expression qui renvoie soit "vrai" soit "faux". Dans cet exemple, la condition est "age >= 18", qui renvoie "vrai" si l'âge est supérieur ou égal à 18 et "faux" sinon.

Gardez à l'esprit que les instructions if peuvent être utilisées seules, mais que les autres instructions ne peuvent pas exister seules. Vous pouvez également créer des conditions plus complexes à l'aide d'opérations mathématiques de base, telles que :

> (plus grand que)

< (moins que)

>= (supérieur ou égal)

<= (inférieur ou égal)

== (équivalent)

UTILISATION DE L'OPERATEUR NOT

Les cas d'utilisation ne nécessitent pas toujours la vérification d'une condition positive, ou vraie, et c'est là que l'opérateur NOT entre en jeu. Représenté par un simple point d'exclamation, l'opérateur NOT permet aux conditions négatives, ou fausses, d'être remplies par les instructions if ou else-if. Cela signifie que les conditions suivantes sont identiques :

```
if(variable == false)
```

ET

```
if(!variable)
```

ÉVALUATION DE CONDITIONS MULTIPLES

Outre l'imbrication des instructions, il est également possible de combiner plusieurs vérifications de conditions en une seule instruction if ou else-if à l'aide des opérateurs logiques AND OR :

L'opérateur ET est représenté par deux caractères `&&`. Toute condition utilisant l'opérateur AND signifie que toutes les conditions doivent être évaluées comme vraies pour que l'instruction if s'exécute.

L'opérateur OR est représenté par deux caractères `||`. Une instruction if utilisant l'opérateur OR s'exécute si une ou plusieurs de ses conditions sont vraies.

Les conditions sont toujours évaluées de gauche à droite.

```
int age = 25;
bool permisDeConduire = true;
if (age >= 18 && permisDeConduire)
{
    Debug.Log("Vous êtes autorisé à conduire !");
}
else if (age >= 16 && !permisDeConduire)
{
    Debug.Log ("Vous pouvez passer le permis de conduire.");
}
else
{
    Debug.Log ("Vous n'êtes pas autorisé à conduire.");
}
```

COMMUTATION AVEC SWITCH

Lorsqu'il y a plus de trois ou quatre cas à traiter à partir d'une expression qui autorise plusieurs valeurs possibles, les instructions switch permettent un format beaucoup plus concis que les instructions if-else.

Correspondance des modèles

Dans les instructions de commutation, la correspondance des motifs fait référence à la manière dont une expression de correspondance est validée par rapport à plusieurs instructions de cas. Une expression de correspondance peut être de n'importe quel type qui n'est pas null ou nothing ; toutes les valeurs de l'instruction case doivent correspondre au type de l'expression de correspondance.

Par exemple, si nous avons une instruction **switch** qui évalue une variable entière, il devrait y avoir autant de bloc **case** que de valeurs différentes. Si ce n'est pas le cas, il faut prévoir le cas par défaut. Dans l'exemple ci-dessous, il est présent mais et inutile car une variable booléenne ne peut prendre que **true** ou **false** comme valeurs : ***MVS le détecte comme code inutile car non accessible***.

```
public class CubeController : MonoBehaviour
{
     public float speed = 5f;

    void Update()
    {
        float moveHorizontal = Input.GetAxis("Horizontal");
        float moveVertical = Input.GetAxis("Vertical");

        Vector3 movement = new Vector3(moveHorizontal, 0f, moveVertical);

        switch (Input.GetKey(KeyCode.Space))
        {
            case true:
                transform.Translate(movement * speed * 2f * Time.deltaTime, Space.World);
                break;
            case false:
                transform.Translate(movement * speed * Time.deltaTime, Space.World);
                break;
            default:
                Debug.Log("Juste pour la synthaxe, c'est impossible");
        }
    }
}
```

Dans cet exemple, nous avons un script **CubeController** qui permet de déplacer un **cube** dans le moteur Unity en utilisant les touches fléchées. Nous utilisons également l'instruction **switch** pour détecter si la touche espace est enfoncée pour accélérer le mouvement.

Dans la méthode Update(), nous utilisons les fonctions `Input.GetAxis("Horizontal")` et `Input.GetAxis("Vertical")` pour détecter les mouvements latéraux et avant/arrière. Nous stockons ensuite ces valeurs dans un vecteur **movement**.

Dans la structure **switch**, nous testons si la touche **espace** est enfoncée en utilisant la fonction `Input.GetKey(KeyCode.Space)`. Si la touche est enfoncée, le code dans le case true sera exécuté, et le cube se déplacera deux fois plus vite que lorsqu'il ne l'est pas. Si la touche n'est pas enfoncée, le code dans le case false sera exécuté et le cube se déplacera normalement.

Dans cet exemple, l'instruction switch est utilisée pour choisir entre deux instructions différentes en fonction de la valeur d'une expression. Dans ce cas, la valeur est un booléen retourné par la fonction `Input.GetKey()`, mais elle pourrait être n'importe quelle expression qui peut être évaluée dans une instruction switch.

TABLEAUX

En C#, un tableau est une structure de données qui permet de stocker plusieurs éléments du même type dans une seule variable. La structure d'un tableau en C# se compose des éléments suivants :

- Type : le type de données que le tableau contient (par exemple, int, string, float, etc.).
- Nom : le nom du tableau qui sera utilisé pour y faire référence dans le code.
- Taille : le nombre d'éléments que le tableau peut contenir.

Voici un exemple de déclaration d'un tableau en C# :

```
int[] tableau = new int[5];
```

Dans cet exemple, nous avons déclaré un tableau nommé tableau qui peut contenir jusqu'à 5 entiers (**int**). Nous avons utilisé le mot clé **new** pour allouer de l'espace en mémoire pour le tableau et initialiser toutes les valeurs à zéro.

Il est également possible d'initialiser les valeurs d'un tableau lors de sa déclaration :

```
int[] tableau = { 1, 2, 3, 4, 5 };
```

Dans cet exemple, nous avons initialisé les valeurs du tableau directement en utilisant une liste d'initialisation. Les accolades délimitent la liste des valeurs à stocker dans le tableau, séparées par des virgules.

INDEXATION ET INDICES

Il est important de noter que les tableaux en C# sont indexés à partir de zéro. Cela signifie que le premier élément d'un tableau a l'indice 0, le deuxième a l'indice 1, et ainsi de suite.

Pour accéder à un élément spécifique dans un tableau, vous pouvez utiliser sa position indexée :

```
int[] tableau = { 1, 2, 3, 4, 5 };
Debug.Log(tableau[2]);
// Affiche "3"
```

TABLEAUX MULTIDIMENSIONNELS

Les tableaux sont également un excellent moyen de stocker des éléments sous la forme d'un tableau (lignes et colonnes dans le monde réel). On parle de tableaux multidimensionnels car chaque élément ajouté apporte une nouvelle dimension aux données. Les exemples de tableaux ci-dessus ne contiennent qu'un élément par index, ils sont donc unidimensionnels.

Si nous voulions qu'un tableau contienne, par exemple, une coordonnée *x* et une coordonnée y dans chaque élément, nous pourrions créer un tableau bidimensionnel comme suit :

```
// Le tableau de coordonnées comporte 3 lignes et 2 colonnes
int[,] Coordinates = new int[3,2] ;
```

Remarquez que nous avons utilisé une virgule à l'intérieur des crochets pour indiquer que le tableau est bidimensionnel et que nous avons ajouté deux champs d'initialisation, qui sont également séparés par une virgule.

Nous pouvons également initialiser directement un tableau multidimensionnel avec des valeurs, en créant un tableau de *x* et de y :

```
int[,] Coordinates = new int[3,2]
{
   {5,4},{1,7},{9,3}
};
```

Dans le code, nous utiliserions l'indice suivant, en commençant par l'indice de la ligne, suivi de l'indice de la ligne. l'index de la colonne :

```
// Recherche de la valeur dans la première ligne, première colonne
int coordinateValue = Coordinates[0, 1] ;
Debug.Log(coordinateValue) ;
```

La modification d'une valeur dans un tableau multidimensionnel se fait de la même manière que pour un tableau ordinaire : nous utilisons l'indice de la valeur que nous voulons mettre à jour, puis nous lui attribuons une nouvelle valeur :

```
// La valeur de la première ligne, première colonne est maintenant 10
Coordonnées [0, 1] = 10 ;
```

Un tableau C# peut avoir jusqu'à 32 dimensions, ce qui est beaucoup, mais les règles pour les créer sont les mêmes - ajouter une virgule supplémentaire pour chaque dimension à l'intérieur des crochets de type au début de la variable et une virgule supplémentaire et le nombre d'éléments dans l'initialisation. Par exemple, un tableau à trois dimensions

```
int[,,] Coordinates = new int[3,3,2] ;
```

Exceptions de portée

Lorsque les tableaux sont créés, le nombre d'éléments est fixé et immuable, ce qui signifie que nous ne pouvons pas accéder à un élément qui n'existe pas.

Tout indice de 3 ou plus est hors de la portée du tableau et génère une erreur

LISTES

Les listes sont étroitement liées aux tableaux :

- On regroupe aussi plusieurs valeurs du même type dans une seule variable
- Elles sont beaucoup plus faciles à gérer lorsqu'il s'agit d'ajouter, de supprimer et de mettre à jour des éléments.
- On peut modifier le nombre d'éléments après la déclaration.
- les listes peuvent être initialisées dans la déclaration de la variable en ajoutant les valeurs des éléments à l'intérieur d'une paire de crochets :

```
List<elementType> name = new List<elementType>() { value1, value2 } ;
```

Un exemple pour ajouter les notes :

```
        List<double> notes = new List<double>();
       // Ajouts de notes avec .Add
        notes.Add(12.5);
        notes.Add(15);
        notes.Add(18.75);
        Debug.Log("Liste des notes :");
       // On fait défiler les éléments pour les afficher une par une
        foreach (double note in notes)
        {
            Debug.Log(note);
        }
       // On calcule la moyenne avec .Average
        double moyenne = notes.Average();
       int nb = notes.Count ;
        Debug.LogFormat ("Moyenne : {0} sur {1} notes", moyenne, nb);
    }
}
```

Remarquez que l'on utilise Count pour les listes, mais Length pour les tableaux afin de connaître le nombre d 'éléments.

Les éléments sont stockés dans l'ordre dans lequel ils sont ajoutés (au lieu de l'ordre séquentiel des valeurs elles-mêmes), sont indexés à zéro comme les tableaux et sont accessibles à l'aide de l'opérateur d'indice.

Connaître le nombre d'éléments d'une liste est très utile ; cependant, dans la plupart des cas, cette information n'est pas suffisante. Nous voulons être en mesure de modifier nos listes en fonction des besoins, ce que nous allons voir par la suite.

Accès et modification des listes

Les éléments d'une liste sont accessibles et modifiables comme des tableaux avec un opérateur d'indice et un index, tant que l'index se trouve dans la plage de la classe Liste. Cependant, la classe Liste dispose d'une variété de méthodes qui étendent ses fonctionnalités, telles que l'ajout, l'insertion et la suppression d'éléments.

Méthodes intéressantes pour la classe List :

- Add() vu dans l'exemple, Insert() pour ajouter ou insérer des éléments :

```
notes.Insert(1, 18) ; // ajoute la note de 18 à l'index 1 et décales les autres éléments.
```

- Suppression

```
QuestPartyMembers.RemoveAt(0) ; // Supprime l'élément à l'index 0
QuestPartyMembers.Remove(18) ; // Supprime la note de 18
```

Si les listes sont idéales pour les éléments à valeur unique, il arrive que l'on ait besoin de stocker des informations ou des données contenant plus d'une valeur. C'est là que les dictionnaires entrent en jeu.

DICTIONNAIRES

Le type **Dictionnaire** s'éloigne des tableaux et des listes en stockant des paires de valeurs dans chaque élément, au lieu de valeurs uniques. Ces éléments sont appelés paires (**clé-valeur**) : la clé sert d'index, ou de valeur de consultation, pour la valeur correspondante. Contrairement aux tableaux et aux listes, les dictionnaires ne sont pas ordonnés. Toutefois, ils peuvent être triés et ordonnés dans diverses configurations après leur création.

La déclaration d'un dictionnaire est presque la même que celle d'une liste, mais avec un détail supplémentaire : les deux la clé et le type de valeur doivent être spécifiés à l'intérieur des symboles de flèche :

```
Dictionary<keyType, valueType> name = new Dictionary<keyType, valueType>() ;
```

Pour saisir les valeurs on procèdera comme suit :

```
Dictionary<keyType, valueType> name = new Dictionary<keyType, valueType>()
{
       {clé1, valeur1},
       {clé2, valeur2}
} ;
```

Les valeurs de clé doivent être unique et ne peuvent pas être modifiées.

Un exemple qui gère un iventaire :

```
// déclaration du dictionnaire
Dictionary<string, int> inventaire = new Dictionary<string, int>();

// Ajouter un article à l'inventaire
inventaire.Add("Pommes", 10);

// Modifier la quantité d'un article dans l'inventaire
inventaire["Pommes"] += 5;

// Supprimer un article de l'inventaire
inventaire.Remove("Pommes");

// Afficher la quantité d'un article dans l'inventaire
if (inventaire.ContainsKey("Pommes"))
{
    Debug.LogFormat("Il reste {0} pommes dans l'inventaire.", inventaire["Pommes"]);
}
else
{
    Debug.LogFormat("L'article 'Pommes' n'est pas dans l'inventaire.");
}
```

Il est préférable d'être certain qu'un élément existe avant d'essayer d'y accéder, afin d'éviter d'ajouter par erreur de nouvelles paires clé-valeur. L'association de la méthode `ContainsKey` à une instruction `if` est la solution la plus simple, puisque `ContainsKey` renvoie une valeur booléenne en fonction de l'existence de la clé :

```
if(inventaire.ContainsKey("Pommes"))
{
       inventaires["Pommes"] = 3 ;
}
```

INSTRUCTIONS D'ITERATION (BOUCLES)

La boucle `for` est le plus souvent utilisée lorsqu'un bloc de code doit être exécuté un certain nombre de fois avant que le programme ne se poursuive. L'instruction elle-même prend en compte trois expressions, chacune ayant une fonction spécifique à exécuter avant que la boucle ne s'exécute. Comme les boucles `for` gardent la trace de l'itération en cours, elles conviennent mieux aux tableaux et aux listes.

En reprenant l'exemple des notes cette fois stockées dans un type `tableau`, on aurait :

```
int[] notes = { 12, 14, 18, 9, 15 }; // déclaration et initialisation du tableau

int somme = 0; // initialisation de la variable somme permettant de cumuler les notes
for (int i = 0; i < notes.Length; i++)
{
    // affichage du n° de la note (i+1) et de la note à l'index i → i est le numéro d'ordre de l'itération en
cours
    Debug.LogFormat ("Note {0} : {1}", i + 1, notes[i]);
    // On ajoute la note d'index i à la variable cumul somme
    somme += notes[i];
}

double moyenne = (double)somme / notes.Length;
Debug.LogFormat ("Moyenne : {0}", moyenne);
```

Dans le même exemple avec les listes, on remarque que la méthode Average permettait de faire plus vite sans besoin d'utiliser une boucle.

En décortiquant la boucle, le mot-clé `for` commence la déclaration, suivi d'une paire de parenthèses. À l'intérieur des parenthèses se trouvent **l'initialisation**, la **condition** et **l'itérateur** expressions.

Dès que la boucle commence, i vaut 0 (initialisation). A la fin de la première boucle, la condition est vérifiée pour savoir si une nouvelle boucle va être exécutée : ici, on aurait i=0 qui est inférieur à notes.length qui est de 5, donc une nouvelle boucle est exécutée avec une valeur de i itérée, c'est à dire que i++ génère i=1.

BOUCLES FOREACH

Nous avons utilisé ci-dessus la boucle `foreach` pour parcourir les notes de la liste nommée **notes** :

```
// On fait défiler les éléments pour les afficher une par une
        foreach (double note in notes)
        {
            Debug.Log(note);
        }
```

Les boucles foreach prennent chaque élément d'une collection et le stockent dans une variable locale, ce qui le rend accessible à l'intérieur de l'instruction. Les boucles foreach peuvent être utilisées avec des tableaux et des listes, mais elles sont particulièrement utiles avec les dictionnaires, étant donné que les dictionnaires sont des paires clé-valeur au lieu d'index numériques.

Sous forme de schéma, une boucle foreach se présente comme suit :

```
Dictionary<string, int> inventaire = new Dictionary<string, int>();
inventaire.Add("Pommes", 10);
inventaire.Add("Oranges", 5);
inventaire.Add("Bananes", 8);

foreach (KeyValuePair<string, int> article in inventaire)
{
    Debug.LogFormat("Article : {0}, Quantité : {1}", article.Key, article.Value);
}
```

BOUCLES "WHILE

Les boucles while sont similaires aux instructions if en ce sens qu'elles s'exécutent tant qu'une seule expression ou condition est vraie.

```
int[] notes = { 12, 14, 18, 9, 15 };
int i = 0;
while (i < notes.Length)
{
    Debug.LogFormat("Note {0} : {1}", i + 1, notes[i]);
    i++;
}
```

Avec les boucles while, il est courant de déclarer une variable d'initialisation, comme dans une boucle for, et de l'incrémenter ou de la décrémenter manuellement à la fin du bloc de code de la boucle. Nous procédons ainsi pour éviter une boucle infinie, dont nous parlerons à la fin du chapitre.

Les boucles while sont très utiles lorsque l'on code en C# , mais elles ne sont pas considérées comme une bonne pratique dans Unity parce qu'elles peuvent avoir un impact négatif sur les performances et qu'elles doivent systématiquement être gérées manuellement.

EXEMPLE AVEC UNE ENUMERATION

```
Dictionary<string, int> inventaire = new Dictionary<string, int>();
inventaire.Add("Pommes", 10);
inventaire.Add("Oranges", 5);
inventaire.Add("Bananes", 8);

IEnumerator<KeyValuePair<string, int>> enumerator = inventaire.GetEnumerator();
while (enumerator.MoveNext())
{
    KeyValuePair<string, int> article = enumerator.Current;
    Debug.LogFormat("Article : {0}, Quantité : {1}", article.Key, article.Value);
}
```

Dans cet exemple, nous avons un script Unity "Inventaire" qui contient un dictionnaire "inventaire" similaire à celui utilisé précédemment. Nous ajoutons des articles à l'inventaire dans la méthode Start().

Nous utilisons ensuite une boucle while pour parcourir chaque élément du dictionnaire. Pour cela, nous appelons la méthode "GetEnumerator()" du dictionnaire qui retourne un objet "IEnumerator" qui permet de parcourir les éléments du dictionnaire un par un. Nous utilisons la méthode "MoveNext()" de l'objet "IEnumerator" pour passer à l'élément suivant du dictionnaire, et nous stockons chaque élément dans un objet "KeyValuePair<string, int>" appelé "article".

TRAVAILLER AVEC DES CLASSES, DES STRUCTURES, ET POO

Les notions qui suivent font partie de la programmation avancée. Jusqu'à présent, nous nous sommes appuyés exclusivement sur des types de variables prédéfinis qui font partie du langage C# : des chaînes de caractères, des listes et des dictionnaires qui sont des classes, ce qui explique pourquoi nous pouvons les créer et utiliser leurs propriétés grâce à la notation par points.

La création de vos classes vous donne la liberté de définir et de configurer des modèles de votre conception, en capturant des informations et en menant des actions spécifiques à votre jeu ou à votre application. Par essence, les classes personnalisées et la POO sont les clés du royaume de la programmation ; sans elles, les programmes uniques seront rares.

Dans ce chapitre, vous allez acquérir une expérience pratique de la création de classes à partir de zéro et discuter du fonctionnement interne des variables de classe, des constructeurs et des méthodes. Vous découvrirez également les différences entre les objets de type référence et de type valeur, et comment ces concepts peuvent être appliqués dans Unity. Les sujets suivants seront abordés plus en détail au fur et à mesure que vous avancerez dans le cours :

- Présentation de la POO
- Définition des classes
- Déclarer des structures
- Comprendre les types de référence et de valeur
- Intégrer l'esprit orienté objet
- Application de la POO dans Unity

PRESENTATION DE LA POO

La programmation orientée objet est le principal paradigme de programmation que vous utiliserez lorsque vous coderez en C# . Si les classes et les structures sont les plans de nos programmes, la POO est l'architecture qui tient le tout ensemble. Lorsque nous parlons de la POO comme d'un paradigme de programmation, nous disons qu'elle a des principes spécifiques sur la façon dont le programme global doit fonctionner et communiquer.

Essentiellement, la POO se concentre sur les objets plutôt que sur la logique séquentielle pure - les données qu'ils contiennent, la façon dont ils conduisent l'action et, surtout, la façon dont ils communiquent entre eux.

DEFINITION DES CLASSES

Nous avons parlé des classes car un script C# est une classe, une classe un peu spéciale car elle est attachée ou basée sur la classe dit MonoBehaviour afin de donner à un script qui est attaché à un objet de disposer de tout un jeu d'instructions qui va permettre au programmeur de gérer le comportement de ses objets.

La principale chose à retenir avec les classes est qu'elles sont de type **référence, c'est-à-dire** que lorsqu'elles sont assignées ou passées à une autre variable, c'est l'objet original qui est référencé, et non une nouvelle copie.

Nous allons aborder la création de classe du point de vue unique de C# en premier lieu.

```
[modificateurs] class NomDeLaClasse [: ClasseDeBase] [, InterfacesImplementees]
{
    //Déclarations de champs, propriétés, méthodes, constructeurs, événements, indexeurs, classes imbriquées et énumérations
}
```

Prenons pour exemple un personnage simple d'un jeu quelconque.

> Créer un nouveau **Script** nommé **Personnage**.
> Supprimer tout le code à l'exception des 3 premières lignes.
> Déclarer la classe Personnage.

On obtient alors le code ci-dessous :

```
using System.Collections ; using System.Collections.Generic ; using UnityEngine ;
public class Personnage
{
}
```

La classe pourrait être ajouté de cette manière dans un projet C# en MVS, seul using UnityEngine serait à retirer.

Le personnage est désormais enregistré en tant que plan de classe publique. Cela signifie que n'importe quelle classe du projet peut l'utiliser pour créer des personnages. Cependant, il ne s'agit que d'instructions - la création d'un personnage nécessite une étape supplémentaire. Cette étape de création est appelée instanciation et fait l'objet de la section suivante.

INSTANCIER DES OBJETS DE CLASSE

L'instanciation est l'acte de créer un objet à partir d'un ensemble spécifique d'instructions, que l'on appelle une **instance**. Si les classes sont des plans, les instances sont les maisons construites à partir de leurs instructions ; toute nouvelle instance de Character est son objet, tout comme deux maisons construites à partir des mêmes instructions restent deux structures physiques différentes. Ce qui arrive à l'une n'a aucune répercussion sur l'autre.

Nous avons créé des listes et des dictionnaires, qui sont des classes par défaut fournies avec C#, en utilisant leurs types et le mot-clé new. Nous pouvons faire la même chose pour les classes personnalisées telles que **Personnage** :

Nous avons déclaré la classe **Personnage** comme publique, ce qui signifie que vous pouvez créer une instance de **Personnage** dans n'importe quelle autre classe.

> Créer un nouveau **Script** nommé **PersonnageTest**.
> Dans **Start**, ajoutez `Personnage hero = new Personnage() ;`

Voyons cela étape par étape :

- Le type de variable est spécifié comme **Personnage**, ce qui signifie que la variable est une **instance** de cette classe.
- La variable s'appelle **hero** et est créée à l'aide du mot-clé `new`, suivi du nom de la classe **Personnage** et de deux parenthèses (). C'est ici que l'instance réelle est créée dans la mémoire du programme, même si la classe est vide pour l'instant.
- Nous pouvons utiliser la variable **hero** comme n'importe quel autre objet avec lequel nous avons travaillé jusqu'à présent. Lorsque la classe **Personnage** aura ses propres variables et méthodes, nous pourrons y accéder à partir de **hero** en utilisant la notation point.

Vous auriez tout aussi bien pu utiliser une déclaration inférée lors de la création de la variable hero, comme suit :

```
var hero = new Personnage();
```

Notre classe de personnage ne peut pas faire grand-chose sans champs de classe. Vous ajouterez des champs de classe et d'autres éléments dans les prochaines sections.

AJOUTER DES CHAMPS DE CLASSE

L'ajout de variables, ou de champs, à une classe personnalisée n'est pas différent de ce que nous avons déjà fait jusqu'à maintenance. Les mêmes concepts s'appliquent, y compris les modificateurs d'accès, la portée des variables et l'attribution de valeurs. Cependant, toutes les variables appartenant à une classe sont créées avec l'instance de la classe, ce qui signifie que si aucune valeur ne leur est attribuée, elles prendront par défaut la valeur **zéro** ou **null**. En général, le choix des valeurs initiales dépend des informations qu'elles stockent :

- Si une variable doit avoir la même valeur initiale à chaque fois qu'une instance de classe est créée, la définition d'une valeur initiale est une bonne idée. Cela peut être utile pour des points d'expériences ou un score de départ.

Si une variable doit être personnalisée dans chaque instance de classe, comme le nom du personnage, laissez sa valeur non assignée et utilisez un constructeur de classe (un sujet que nous aborderons plus loin).

Chaque classe de personnage aura besoin de quelques champs de base ; c'est à vous de les ajouter : Incorporons deux variables pour contenir le nom du personnage et le nombre de points d'expérience de départ :

> Ajoutez une variable de type **string** en accès **public** à l'intérieur des accolades de la classe **Personnage**.
> Ajoutez une variable de type entier pour les points d'expérience.
> Laissez la valeur du nom vide, mais fixez les points d'expérience à 0 afin que chaque personnage commence au bas de l'échelle :

```
public class Personnage
{
    public string nom;
    public int exp = 0;
}
```

> Dans le script **PersonnageTest**, juste après l'initialisation de l'instance de personnage, affichons les valeurs qui ont été affectées aux 2 champs qui décrivent notre personnage nommé hero. :
> `Debug.LogFormat("Le personnage nommé Hero : {0} - {1} EXP", hero.nom, hero.exp) ;`

Lorsque l'objet **hero** est initialisé, le nom se voit attribuer une valeur nulle qui apparaît comme un espace vide dans le journal de débogage, tandis que l'exp s'imprime comme 0.

> ***Remarquez que nous n'avons pas eu à attacher le script** Personnage **à des GameObjects dans la scène ; nous les avons simplement référencés dans** PersonnageTest **et Unity s'est chargé du reste.***

À ce stade, notre classe fonctionne, mais elle n'est pas très pratique avec ces valeurs vides : c'est le rôle des constructeurs d'initialiser ce qui est important lors de l'instanciation au niveau de la classe.

UTILISATION DES CONSTRUCTEURS

Les constructeurs de classe sont des méthodes spéciales qui se déclenchent automatiquement lors de la création d'une instance de classe, de la même manière que la méthode Start s'exécute dans un script basé sur MonoBehaviour.

Les constructeurs construisent la classe conformément à son plan :

Si aucun constructeur n'est spécifié, C# en génère un par défaut. Le constructeur par défaut donne à toutes les variables la valeur de leur type par défaut : les valeurs numériques sont fixées à 0, les booléens à false et les types de référence (classes) à null.

Les constructeurs personnalisés peuvent être définis avec des paramètres, comme n'importe quelle autre méthode, et sont utilisés pour définir les valeurs des variables de classe lors de l'initialisation.

Une classe peut avoir plusieurs constructeurs.

Les constructeurs s'écrivent comme des méthodes ordinaires, à quelques différences près : par exemple, ils doivent être publics, n'ont pas de type de retour et le nom de la méthode est toujours le nom de la classe.

À titre d'exemple, ajoutons un constructeur de base sans paramètre à la classe **Personnage** et attribuons au champ nom une valeur autre que **null**.

> Ajoutez ce nouveau code directement sous les variables de la classe, comme suit :

```
public Personnage()
{
    name = "Non assigné";
}
```

> Lancez le projet dans Unity et vous verrez l'instance utiliser ce nouveau constructeur.

C'est un bon progrès, mais nous avons besoin que le constructeur de la classe soit plus flexible. Cela signifie que nous devons pouvoir passer des valeurs afin qu'elles puissent être utilisées comme valeurs de départ, ce que vous ferez par la suite.

La classe **Personnage** commence à se comporter comme un véritable objet, mais nous pouvons encore l'améliorer en ajoutant un second constructeur qui prend un nom lors de l'initialisation et le place dans le champ **nom** :

> Ajoutez un autre constructeur à **Personnage** qui prend un paramètre de type **string**, appelé **nom**. Le fait d'avoir plusieurs constructeurs dans une même classe s'appelle la surcharge des constructeurs.
>
> Attribuer le paramètre à la variable nom de la classe à l'aide du mot clé **this** : le code qui en résulte est récapitulé ci-dessous :

```
public class Personnage
{
    public string nom;
    public int exp = 0;
    public Personnage()
    {
        nom = "Non assigné";
    }
    public Personnage(string nom)
    {
        this.nom = nom;
    }
}
```

Pour des raisons de commodité, les constructeurs ont souvent des paramètres qui partagent un nom avec une variable de classe. Dans ce cas, il convient d'utiliser le mot-clé **this** pour spécifier quelle variable appartient à la classe. Dans l'exemple ci-dessus, `this.nom` fait référence à la variable nom de la classe, tandis que nom est le paramètre. Sans le mot-clé **this**, le compilateur émettra un avertissement car il ne sera pas en mesure de les différencier. Pour plus de clarté, vous auriez également pu utiliser le mot-clé **this** dans le constructeur par défaut où nous avons défini la propriété nom e à « non assigné ».

> Testez par vous-même et retirer this. Devant nom dans le deuxième constructeur.
>
> Créer une nouvelle instance de personnage dans **PersonnageTest**, appelée heroine :
> ```
> Personnage heroine = new Character("Agatha") ;
> Debug.LogFormat("Hero : {0} - {1} EXP", heroine.nom, heroine.exp) ;
> ```

Lorsqu'une classe possède plusieurs constructeurs ou qu'une méthode possède plusieurs variantes, MVS propose dans la fenêtre contextuelle d'auto-complétions 2 choix., que l'on peut faire défiler à l'aide des touches fléchées afin de voir les constructeurs existants dans la classe.

DECLARER LES METHODES DE LA CLASSE

Ajouter des méthodes à des classes personnalisées n'est pas différent de les ajouter à un script normal. Cependant, c'est une bonne occasion de parler d'un élément fondamental de la bonne programmation - ne pas se répéter (DRY). DRY est une référence pour tout code bien écrit. Essentiellement, si vous vous retrouvez à écrire la même ligne, ou les mêmes lignes, encore et encore, il est temps de repenser et de réorganiser. Cela prend généralement la forme d'une nouvelle méthode pour contenir le code répété, ce qui facilite la modification et l'appel de cette fonctionnalité ailleurs dans le script en cours ou même à partir d'autres scripts.

En termes de programmation, on parle d'abstraction d'une méthode ou d'une fonctionnalité.

Nous avons déjà pas mal de code répété, alors jetons un coup d'œil et voyons où nous pouvons augmenter le nombre d'heures de travail. la lisibilité et l'efficacité de nos codes.

> ***Les instructions** Debug.Log **dans le code sont répétés 2 fois, il est donc pertinent de créer une méthode pour afficher les statistiques du personnage.***

De plus, nous comprenons bien que si dans l'immédiat nous n'avons pas envie d'aller plus loin dans l'aspect design de l'affichage des stats, on va essayer d'illustrer l'importance de séparer les fonctionnalités afin de re-développer (refactoring) l'affichage via une jolie UI.

Cette méthode n'a pas besoin de retourner une valeur.

Elle n'a pas besoin de paramètres car les champs sont accessibles directement dans la classe.

> Dans la classe **Personnage**, on peut ajouter le code ci-dessous pour notre méthode nommée `AfficheStats` :

```
public virtual void AfficheStats()
    {
        Debug.LogFormat("Statistiques du personnage : {0} - {1} EXP", this.nom, this.exp);
    }
```

> Dans la classe **PersonnageTest**, mettez en commentaire le code dans start() comme suit :

```
        /* Version 1
        // On instancie l'objet héro de la classe Personnage
        Personnage hero = new Personnage();
```

```
        // On affiche ses champs
        Debug.LogFormat("Hero : {0} - {1} EXP", hero.nom, hero.exp);
        Personnage heroine = new Personnage("Agatha");
        Debug.LogFormat("Heroine : {0} - {1} EXP", heroine.nom, heroine.exp);
         */
```

De cette façon, notre code est conservé pour mémoire en version 1

On peut maintenant ajouter une version 2 légèrement différente en 2 phases : 1 pour instancier nos 2 objets et 1 pour gérer l'affichage qui est plus utile à du débogage qu'à l'application elle-même.

> Intégrez la méthode `AfficheStats` :à la place de `Debug.LogFormat` :

```
        // Version 2
        // Instanciation
        Personnage hero = new Personnage();
        Personnage heroine = new Personnage("Agatha");
        // Affichage
        hero.AfficheStats();
        heroine.AfficheStats();
```

N'importe quelle instance peut accéder librement à la méthode AfficheStats(). Si une modification doit être faite concernant l'affichage, celle-ci le sera 1 et 1 seule fois.

On retient qu'une méthode est utile pour ajouter une fonctionnalité, rendre le code plus lisible, éviter de répéter le code.

DECLARER DES STRUCTURES

Les structures sont similaires aux classes en ce sens qu'elles sont également des plans pour les objets que vous souhaitez créer dans vos programmes. La principale différence réside dans le fait qu'il s'agit de **types de valeur**, ce qui signifie qu'ils sont transmis par valeur et non par référence, comme le sont les classes. Lorsque les structures sont assignées ou passées à une autre variable, une nouvelle copie de la structure est créée, de sorte que **l'original est déconnecté complètement de la copie**.

Les structures sont déclarées de la même manière que les classes et peuvent contenir des champs, des méthodes et des constructeurs : l'exemple que l'on va prendre sera appliqué à la gestion des armes de nos personnages (la structure est préférable à une classe ici, mais nous y reviendrons plus loin).

> Créez un script nommé **Arme**.
> Comme pour la classe Personnage, notre script n'est pas basé sur **MonoBehaviour** : retirez le code inutile.
> Déclarez une structure en accès **public** appelée **Arme**.
> Ajouter un **champ** pour le **nom** de type **string** et un autre champ pour le **dommage** de type **int**.
> Ajouter le **Constructeur** (comme pour la classe **Personnage** mais celui-ci doit avoir des arguments).
> Comme pour la classe, ajoutez une méthode AfficheStats() pour contrôler les valeurs des champs.
> Dans le script **PersonnageTest**, ajoutez **l'instanciation** avec **marteau** comme nom d'objet d'une arme (nom = « Marteau » et dommage = 105).

Le résultat doit être :

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public struct Arme
{
    public string nom;
    public int dommage;

    public Arme(string nom, int dommage)
    {
        this.nom = nom;
        this.dommage = dommage;
    }
    public void AfficheStats()
    {
        Debug.LogFormat("Arme: {0} - {1} DMG", this.nom, this.dommage);
     }
}
```

Pour le script, les lignes à insérer sont :

```
// Instanciation
Arme marteau = new Arme("Marteau", 105);
// Affichage
marteau.AfficheStats();
```

Les classes et les structures peuvent être imbriquées les unes dans les autres, mais cette pratique est généralement déconseillée car elle encombre le code :

Comme les classes, les variables et les méthodes appartiennent exclusivement à la structure et sont accessibles par son nom unique.

Cependant, les structures ont quelques spécificités :

- Les variables ne peuvent pas être initialisées avec des valeurs à l'intérieur de la déclaration **struct** à moins qu'elles ne soient marquées avec le modificateur **static** ou **const**
- Les constructeurs sans paramètres ne sont pas autorisés.
- Les structures sont dotées d'un constructeur par défaut qui fixe automatiquement toutes les variables à leur valeur par défaut en fonction de leur type.

COMPRENDRE LES TYPES DE REFERENCE ET DE VALEUR

Les classes sont des **types de référence**, c'est-à-dire qu'elles sont transmises par référence ; les structures sont des **types de valeur**, c'est-à-dire qu'elles sont transmises par valeur.

TYPES DE REFERENCE

Lorsque les instances de notre classe Personnage sont initialisées, les variables hero et heroine ne contiennent pas d'informations sur leur classe, mais une référence à l'emplacement de l'objet dans la mémoire du programme. Si nous assignons hero ou heroine à une autre variable de la même classe, c'est la référence mémoire qui est assignée, et non les données du personnage. Cela a plusieurs implications, la plus importante étant que si nous avons plusieurs variables stockant la même référence mémoire, une modification de l'une d'entre elles les affecte :

> Ajoutez le code ci-dessous en version 3 dans le script PersonnageTest :

```
// Version 3
Personnage hero = new Personnage();
Personnage vilain = hero;
hero.AfficheStats();
vilain.AfficheStats();
```

Les deux journaux de débogage seront identiques car vilain a été assigné à hero lors de sa création. À ce stade, hero et villain pointent tous deux vers l'endroit où hero est stocké en mémoire :

Maintenant, changez le nom du méchant et réaffichez les stats an ajoutant les lignes ci-dessous :

```
vilain.nom = "Cruella";
hero.AfficheStats();
vilain.AfficheStats();
```

On voit bien que les deux personnages ont le même nom (« Cruella ») bien que l'on a modifié à priori seulement le nom du vilain

On retiendra que la copie d'un type « par référence » est délicate et que la copie à des conséquences pouvant être surprenantes : c'est en partie pour cette raison que le passage des paramètres/arguments dans une méthode est par défaut un passage par valeur et non par référence et il est donc obligatoire d'ajouter le mot-clé **ref.**

Si vous essayez de copier une classe, vous devez soit créer une nouvelle instance distincte, soit reconsidérer la question de savoir si une structure ne serait pas un meilleur choix pour votre modèle d'objet. Vous aurez un meilleur aperçu des types de valeurs dans la section suivante.

TYPES DE VALEURS

Lorsqu'un objet **struct** est créé, toutes ses données sont stockées dans la variable correspondante, sans référence ni connexion à son emplacement en mémoire. Les structures sont donc utiles pour créer des objets qui doivent être copiés rapidement et efficacement, tout en conservant leur identité propre.

Essayez ce code :

```
// Version 4
Arme marteau = new Arme("Marteau", 105);
Arme lance = marteau;
marteau.AfficheStats();
lance.AfficheStats();
lance.nom = "Lance";
marteau.AfficheStats();
lance.AfficheStats();
```

On voit bien que les données modifiées sont en cohérence avec une logique qui est peut-être plus naturelle : on copie l'objet en dupliquant toutes ses champs/propriétés puis on modifie seulement ceux qui doivent l'être. L'original n'est pas modifié.

Le but, bien entendu n'est pas d'opposer les deux, mais de commencer à admettre que le passage de données par référence ou par valeur est une notion importante dans la programmation.

INTEGRER L'ESPRIT ORIENTE OBJET

Les objets du monde physique fonctionnent à un niveau similaire à celui de la POO ; lorsque vous voulez acheter une boisson gazeuse, vous prenez une canette de soda, et non le liquide lui-même. La canette est un objet, qui regroupe des informations et des actions connexes dans un ensemble autonome. Cependant, il existe des règles concernant les objets, tant en programmation qu'à l'épicerie - par exemple, qui peut y avoir accès. Les différentes variations et les actions génériques jouent toutes un rôle dans la nature des objets qui nous entourent.

En termes de programmation, ces règles sont les principaux principes de la POO : l'encapsulation, l'héritage et l'utilisation de l'information. le polymorphisme. Nous allons aborder ces sujets dans les prochaines sections !

ENCAPSULATION

L'un des avantages de la POO est qu'elle permet l'encapsulation, c'est-à-dire la définition de l'accessibilité des variables et des méthodes d'un objet au code extérieur (ce que l'on appelle parfois le code d'appel). Prenons l'exemple de notre canette de soda : dans un distributeur automatique, les interactions possibles sont limitées. Comme la machine est verrouillée, n'importe qui ne peut pas s'approcher et en prendre une ; si vous avez la bonne monnaie, vous aurez un accès provisoire à la canette, mais dans une quantité spécifiée. Si la machine elle-même est enfermée dans une pièce, seule la personne possédant la clé de la porte saura que la canette de soda existe.

Comment fixer ces limites ? La réponse est simple : nous avons toujours utilisé l'encapsulation en spécifiant des modificateurs d'accès pour les variables et les méthodes de nos objets.

Essayons un exemple simple d'encapsulation pour comprendre comment cela fonctionne en pratique. Notre classe Personnage est déclaré en public, tout comme ses champs et ses méthodes. Cependant, que se passerait-il si nous voulions une méthode capable de réinitialiser les données d'un personnage à leurs valeurs initiales ? Cette méthode pourrait s'avérer utile, mais elle pourrait s'avérer désastreuse si elle était appelée accidentellement, ce qui en fait un candidat parfait pour un membre d'objet privé :

> Créez une méthode privée appelée Reset, sans valeur de retour, dans la classe Personnage.
> Spécifiez respectivement les variables nom et exp à "Non assigné" et 0 :

```
private void Reset()
{
    this.nom = "Non attribué"; this.exp = 0;
}
```

Essayez d'appeler Reset() à partir du script **PersonnageTest** :

```
// Version 5
Personnage hero2 = new Personnage("Lancelot");
hero2.AfficheStats();
hero2.Reset();
```

MVS vous informe que Reset est inaccessible en raison de son niveau de protection.

Si cependant on appelle Reset à l'intérieur de la classe Personnage, dans le premier constructeur par exemple comme suit :

```
public Personnage()
{
    //nom = "Non attribué";
    Reset();
}
```

Dans ce cas, MVS ne déclenche aucun problème de compilation : la méthode Reset(), si elle est problématique en appel extérieur à la classe Personnage, peut servir à remettre à zéro des valeurs à plusieurs endroits dans la classe.

HERITAGE

Une classe C# peut être créée à l'image d'une autre classe, en partageant ses variables membres et ses méthodes, mais en étant capable de définir ses propres données. En POO, on parle d'**héritage, et c'est** un moyen puissant de créer des classes apparentées sans avoir à répéter le code. Reprenons l'exemple des sodas : il existe sur le marché des sodas génériques qui possèdent toutes les mêmes propriétés de base, puis des sodas spéciaux. Les sodas spéciaux partagent les mêmes propriétés de base, mais leur marque, ou leur emballage, les différencie. Lorsque vous les regardez côte à côte, il est évident qu'il s'agit de deux canettes de soda, mais il est tout aussi évident qu'elles ne sont pas identiques.

La classe d'origine est généralement appelée classe de base ou classe mère, tandis que la classe qui en hérite est appelée classe dérivée ou classe enfant. Tous les membres de la classe de base marqués par les modificateurs d'accès public, protégé ou interne font automatiquement partie de la classe dérivée, à l'exception des constructeurs. Les constructeurs de classe appartiennent toujours à la classe qui les contient, mais ils peuvent être utilisés à partir des classes dérivées afin de réduire au minimum la répétition du code.

La plupart des jeux ont plus d'un type de personnage, on peut alors imaginer une nouvelle classe appelée Paladin qui hérite de la classe Personnage. Vous pouvez ajouter cette nouvelle classe au script Personnage ou en créer un nouveau. Si vous ajoutez la nouvelle classe au script Personnage, assurez-vous qu'elle se trouve en dehors des crochets de la classe Personnage :

```
    public class Personnage
    {
        // Tout notre code précédent...
    }
    public class Paladin : Personnage
    {
    }
```

De la même manière que PersonnageTest hérite de MonoBehaviour, les instances de Paladin auront accès aux propriétés nom et exp, ainsi ainsi qu'à la méthode PrintAfficheStats().

CONSTRUCTEURS DE BASE

Lorsqu'une classe hérite d'une autre classe, elle forme une sorte de pyramide dont les variables membres descendent de la classe mère vers tous ses enfants dérivés. La classe mère ne connaît aucun de ses enfants, mais tous les enfants connaissent leur parent. Cependant, les constructeurs de la classe mère peuvent être appelés directement à partir des constructeurs de la classe enfant, moyennant une simple modification de la syntaxe :

```
public class ClasseEnfant : ClasseParent
{
        public ClasseEnfant() : base()
        {
        }
}
```

Le mot-clé base représente le constructeur parent - dans ce cas, le constructeur par défaut. Cependant, puisque base représente un constructeur et qu'un constructeur est une méthode, une classe enfant peut passer des paramètres à son constructeur parent en remontant la pyramide.

Puisque nous voulons que tous les objets Paladin aient une variable nom, et que Personnage a déjà un constructeur qui gère cela, nous pouvons appeler le constructeur de base directement depuis la classe Paladin et nous épargner la réécriture d'un constructeur en passant par exemple le paramètre nom :

```
public class Paladin : Personnage
{
   public Paladin(string nom) : base(nom)
   {
   }
}
```

> Dans **Personnage** , créez une nouvelle instance de Paladin appelée **chevalier**
> Utilisez le constructeur de base pour assigner une valeur à **nom**.
> Appelez `AffichetStats` à partir de **chevalier** et regardez la console :
>
> ```
> Paladin chevalier = new Paladin("Sir Arthur") ;
> chevalier.AfficheStats() ;
> ```

Lorsque le constructeur Paladin se déclenche, il transmet le paramètre nom au constructeur Personnage, qui définit la valeur de nom. Le constructeur Personnage est utilisé pour initialiser la classe Paladin, rendant le constructeur Paladin uniquement responsable de l'initialisation de ses propriétés uniques, qu'il n'a pas à ce stade.

Outre l'héritage, il peut arriver que vous souhaitiez créer de nouveaux objets à partir d'une combinaison d'autres objets existants. Pensez aux LEGO® ; vous ne commencez pas à construire à partir de rien - vous avez déjà des blocs et des structures de différentes couleurs avec lesquels vous pouvez travailler. En termes de programmation, c'est ce qu'on appelle la composition, dont nous parlerons dans la section suivante.

COMPOSITION

Outre l'héritage, les classes peuvent être composées par d'autres classes. Prenons l'exemple de notre structure Arme. Paladin peut facilement contenir une variable Arme à l'intérieur de lui-même et avoir accès à toutes ses propriétés et méthodes. Pour ce faire, nous allons mettre à jour Paladin pour qu'il prenne en charge une arme de départ et lui assigner sa valeur dans le constructeur :

```
public class Paladin : Personnage
{
        Public Arme arme ;
        public Paladin(string nom, Arme arme) : base(nom)
        {
           this.arme = arme ;
        }
}
```

Comme l'arme est propre au Paladin et non au personnage, nous devons définir sa valeur initiale dans le constructeur. Nous devons également mettre à jour l'instance du chevalier pour inclure une variable Arme. Dans PersonnageTest, ajoutons la ligne ci-dessous pour déclarer que le chavalier va disposer d'un marteau : `Paladin chevalier = new Paladin("Sir Arthur", marteau) ;`

Si vous lancez le jeu maintenant, vous ne verrez rien de différent parce que nous utilisons la méthode `AfficheStats()` de la classe **Personnage**, qui ne connaît pas la propriété **Arme** de la classe **Paladin**. Pour résoudre ce problème, nous devons parler de **polymorphisme**.

POLYMORPHISME

Le polymorphisme est le mot grec qui signifie « formes différentes » et s'applique à la POO de deux manières distinctes :

- Les objets de classe dérivés sont traités de la même manière que les objets de classe parents. Par exemple, un tableau d'objets Personnage peut également contenir des objets Paladin, puisqu'ils dérivent de Personnage.
- Les classes mères peuvent marquer les méthodes comme virtuelles, ce qui signifie que leurs instructions peuvent être modifiées par les classes dérivées à l'aide du mot-clé override. Dans le cas de Personnage et Paladin, il serait utile de pouvoir déboguer différents messages de `AfficheStats` pour chacun d'entre eux.

Le polymorphisme permet aux classes dérivées de conserver la structure de leur classe mère tout en ayant la liberté d'adapter les actions à leurs besoins spécifiques. Toute méthode que vous marquez comme virtuelle vous donnera la liberté du polymorphisme d'objet. Prenons ces nouvelles connaissances et appliquons-les à notre méthode de débogage de personnage.

Modifions **Personnage** et Paladin pour qu'ils affichent différents journaux de débogage à l'aide de `AfficheStats` :

Modifier `AfficheStats` dans la classe **Personnage** en ajoutant le mot-clé **virtual** entre **public** et **void** :

```
public virtual void AfficheStats()
{
   Debug.LogFormat("Héros : {0} - {1} EXP", nom, exp) ;
}
```

1. Déclarez la méthode `AfficheStats` dans la classe **Paladin** en utilisant le mot-clé **override**.
2. Ajoutez un journal de débogage pour imprimer les propriétés de Paladin de la manière que vous souhaitez :

```
public override void AfficheStats ()
{
   Debug.LogFormat("Salut {0} - prenez votre {1} !", this.nom, this.arme.nom) ;
}
```

> ***Cela peut ressembler à du code répété, ce qui, nous l'avons déjà dit, n'est pas une bonne chose, mais il s'agit d'un cas particulier. Ce que nous avons fait en marquant `AfficheStats` comme virtuel dans la classe Personnage, c'est d'indiquer au compilateur que cette méthode peut avoir plusieurs formes en fonction de la classe appelante.***

Lorsque nous avons déclaré la version surchargée de `AfficheStats` dans **Paladin**, nous avons ajouté le comportement personnalisé qui ne s'applique qu'à cette classe. Grâce au **polymorphisme**, nous n'avons pas à choisir quelle version de `AfficheStats` nous voulons appeler à partir d'un objet **Personnage** ou **Paladin** - le compilateur le sait déjà :

Ces notions sont loin de devoir être considérées comme simples et évidentes, c'est petit à petit en fonction des besoins de conception que l'on en voit l'utilité, la puissance via une maintenance facilitée du code. On peut continuer à apprendre en assimilant petit à petit ces notions quand elles apparaissent dans différents exemples.