Table des matières

PROGRAMMATION EN C

- Le langage C
- Le langage C - Les bases de la syntaxe en C
- Python - C: les différences de syntaxe
- Écrire du C - Code::Blocks IDE
- C - Les variables
- C - Les types de variables
- C - Les nombres entiers (`int`)
- C - Les nombres à virgule (`float` et `double`)
- C - Les caractères (`char`)
- C - Assigner une valeur à des strings
- C - Les booléean (`bool`)
- 💪 C - Les variables (Exercices)
- C - Opérations mathématiques
- C - Les noms de variables
- C - Affichage avec `printf`
- 💪 C - `printf` exercices
- C - Lecture au clavier avec `scanf`
- C - Les commentaires
- C - Les structures (`struct`)
- C - Les conditions
- 💪 C - Les conditions (exercices)
- C - Les fonctions
- C - Passage de paramètres par valeur et par référence
- C - Les fonctions - Exercices 💪
- C - Comment documenter correctement une fonction
- C - Les boucles
- C - Les boucles `for`
- C - Les boucles `for` - Exercices 💪
- C - #include

© 2025 Mr. Lambrechts

C - #include - TP 💪

C - Créateur de joueur de foot 🏆

PROGRAMMATION EN PYTHON

Python

Tester ses programmes en Python

pytest - bonnes pratiques

pytest - exemple complet

Tester ses programmes en Python - Exercices ✍️

Python: Les boucles

Python: Les boucles while

Python: Les boucles while (exercices)

Python: Les boucles while - Validation

Python: Les boucles for imbriquées

Python: Les boucles for

Python: Compter et cumuler des valeurs dans une boucle

Python: Les boucles for sur des listes

Python: Afficher un menu de sélection

Python: afficher un labyrinthe

Python: Les modules

ELECTRONIQUE

Les capteurs

BME280: pression atmosphérique et altitude

DHT11: Simulation du Capteur

DHT11: Capteur de température et d'humidité

HC-SR04: Capteur de distance à ultra-sons

Les micro-contrôleurs

Le Raspberry Pi Pico

Les Broches du Raspberry Pi Pico en MicroPython

MicroPython sur RPi Pico avec Thonny Python

Raspberry Pi Pico et les Ports GPIO

Utilisation du Raspberry Pi Pico avec MicroPython

Les différents types de signaux

INTELLIGENCE ARTIFICIELLE

Chatbot, assistant intégré, agent : qui fait quoi ?

Exercice — Coder son projet avec un agent IA

INTERNET

Création de pages Web

Javascript

💪🏻 JavaScript - Boucles for - Exercices

🎯 Bases de Javascript: Carnet d'objectifs

JavaScript - La console développeur

JavaScript - Syntaxe et variables

JavaScript - Exercices de base

JavaScript - La saisie utilisateur

JavaScript - Les conditions

JavaScript - Combiner plusieurs conditions

JavaScript - La boucle for

JavaScript - La boucle while

JavaScript - Les fonctions

JavaScript - Manipuler le DOM

JavaScript - Manipuler le style et les images avec JavaScript

JavaScript - Manipuler le style et les images (exercices 🚲)

JavaScript - Les formulaires

JavaScript - Réagir aux évènements

JavaScript - Tableaux, objets et JSON

JavaScript - Projet intégrateur

Création de pages Web avec Python

Flask - Les routes

Flask - Les routes | Exercices

Flask - Les routes avec variables

Flask - Les fichiers statiques

Flask - url_for
Jinja2 - Templates - Héritage
Introduction à Jinja2
Projets Flask
Insertion des données (initial load)
Intégration d'un template existant

SYSTÈMES D'EXPLOITATION

Fichiers et dossiers
Organisez votre vidéothèque (2025)
Les outils Windows
Total Commander
TC - Multi-Rename Tool
TC - Synchronize Dirs
TC - 25 raccourcis clavier essentiels
Windows

OUTILS INDISPENSABLES

Markdown: écrire du texte clair et structuré
Markdown: convertir en HTML avec Python
Markdown: conversion HTML en batch
Markdown: conversion HTML en batch, avec template

ARCHITECTURE LOGICIELLE

Découvrir l'architecture logicielle
Découpler tes composants
Éviter les nombres magiques dans le code

CANSAT

Liens utiles
Cansat - Projets paralèlles

SYSTÈMES INFORMATIQUES

Codage de l'information

Système de Numération et le Système Décimal

Pourquoi utiliser le complément à 2

Représenter les nombres signés

Représentation des Chaînes de Caractères en Informatique

PROGRAMMATION ORIENTÉE OBJET

OOP - Étude de cas D&D

Chapitre 1 – Classes et objets

Exercices – Classes et objets

Chapitre 2 – Attributs et visibilité

Chapitre 3 – Getters et setters

Chapitre 4 – Les propriétés C#

Chapitre 5 – Les méthodes

Chapitre 6 – Le constructeur

GESTION DE PROJETS

Le carré des contraintes

Méthodes agiles

Les User Stories

A rétrospective

La méthode Kanban

Le Daily Stand-Up

Questionnaire

Make/Buy/Reuse (MBR)

La matrice Urgent / Important

Les Time-Targets

La matrice Valeur / Effort

La méthode MoSCoW

Les types de bugs et leur gravité

Projet d'examen — Système multi-langages (C · Python · Flask · C#)

LE LANGAGE C-SHARP (C#)

- Mise en place de l'environnement C#
- La syntaxe de base du C#
- Exercice — Todo List en console
- Exercice — Générateur de mot de passe (app fenêtrée)
- Les conventions de nommage en C#



PROGRAMMATION EN C

Le langage C

> *Voici les choses essentielles à connaître sur le langage C.*

Objectifs

À la fin de ce chapitre, tu sauras :

- Expliquer la différence entre un langage **compilé** et un langage **interprété**.
- Comprendre pourquoi on apprend encore le C aujourd'hui.
- Connaître les outils nécessaires pour programmer en C.

Le C en quelques mots

Le langage C a été créé dans les années 1970.

- Il a servi à écrire **le système Unix** et plus tard **Linux**.
- Beaucoup de logiciels systèmes (compilateurs, bases de données, microcontrôleurs) utilisent encore le C.
- Le C est un langage **rapide** et **"proche" de la machine** : il permet de comprendre comment l'ordinateur gère la mémoire et exécute les instructions.

Langages compilés et interprétés

Il existe deux grandes familles de langages :

- **Langages interprétés** (ex. Python, JavaScript)
  - Ton code est **lu ligne par ligne** par un interpréteur.
  - Tu dois avoir cet interpréteur installé pour lancer ton programme.
    - Exemple : `python moncode.py` → il faut Python installé.
  - Avantage : facile à tester rapidement.
  - Inconvénient : plus lent, dépend toujours de l'interpréteur.
- **Langages compilés** (ex. C, C++)
  - Ton code est **traduit en langage machine** avant d'être exécuté → création d'un fichier exécutable.
  - Une fois compilé, ton programme peut tourner **sans avoir besoin du compilateur**.
    - Exemple : `moncode.c` → compilation → `moncode.exe`, exécutable directement.
  - Avantage : programme **rapide et autonome**.

- Inconvénient : il faut passer par l'étape de **compilation**.

💡 **Comparaison rapide :**

- Python → `python moncode.py` (l'interpréteur est indispensable).
- C → `gcc moncode.c -o moncode` puis `moncode.exe` (le programme compilé peut être exécuté seul).

Pourquoi apprendre le C

- Comprendre **comment fonctionne un ordinateur** en profondeur.
- Apprendre la rigueur d'un langage plus « strict » que Python ou JavaScript.
- Utile pour la **programmation système**, l'**embarqué** (microcontrôleurs, robots), et même pour certains moteurs de jeux vidéo.

Les outils à utiliser

Pour écrire et lancer un programme en C, tu as besoin de :

1. **Un éditeur de code / IDE** : Code::Blocks, Visual Studio Code, CLion…
2. **Un compilateur** : le plus courant est `gcc` (GNU Compiler Collection).
3. **Un terminal / console** (éventuellement) pour compiler et exécuter ton programme.

👉 On fera un article spécifique pour t'expliquer **comment installer et utiliser ces outils**.

🔥 Compilateur: logiciel/programme qui traduit un code source en langage machine exécutable par l'ordinateur.

Exercices de réflexion

1. Classe ces langages dans la bonne catégorie : C, JavaScript, Python, C++, C#.
2. Pourquoi dit-on qu'un programme compilé est plus rapide qu'un programme interprété ?
3. Imagine : tu veux programmer un petit jeu sur ton navigateur. Choisirais-tu plutôt C ou JavaScript ? Pourquoi ?
4. Explique pourquoi tu ne peux pas lancer directement `moncode.c` comme un fichier exécutable.

À retenir

- Le C est un langage **compilé**, contrairement à Python ou JavaScript qui sont **interprétés**.
- Un langage interprété → nécessite **toujours l'interpréteur installé** pour exécuter ton programme.
- Un langage compilé → produit un **exécutable autonome**, que tu peux lancer directement.
- Le C reste aujourd'hui incontournable pour comprendre **l'informatique au plus près de la machine**.

Le langage C - Les bases de la syntaxe en C

> *Voici les choses essentielles à connaître sur la syntaxe du C.*

Objectifs

À la fin de ce chapitre, tu sauras :

- Comprendre la structure minimale d'un programme en C.
- Exécuter ton premier programme avec `printf`.
- Expliquer le rôle des accolades `{}` et du point-virgule `;`.
- Utiliser des commentaires pour documenter ton code.
- Compiler et exécuter un programme en C.

Structure minimale d'un programme

En C, tout programme contient **au minimum** :

```c
#include <stdio.h>   // bibliothèque pour afficher du texte

int main() {
    return 0; // indique que le programme s'est bien terminé
}
```

- `#include <stdio.h>` : permet d'utiliser la fonction `printf`.
- `int main() { ... }` : le point d'entrée du programme (là où tout commence).
- `return 0;` : dit au système que le programme s'est terminé correctement.

---

2 Les instructions et les blocs

- Une instruction se termine toujours par un **point-virgule** `;`.
- Les **accolades** `{ }` regroupent plusieurs instructions en bloc (ex. dans `main`).

Sans le `;` ou sans les `{}`, ton programme ne compilera pas.

Les commentaires

Un commentaire sert à expliquer ton code. Le compilateur les ignore.

- Une ligne :

```
// Ceci est un commentaire
```

- Plusieurs lignes :

```
/* Ceci est
   un commentaire
   sur plusieurs lignes */
```

Compilation et exécution

Contrairement à Python, le C **doit être compilé** avant de s'exécuter.

- Étape 1 : Écrire ton fichier `hello.c`.
- Étape 2 : Compiler avec `gcc hello.c -o hello`
- Étape 3 : Lancer le programme avec `./hello`

Exemple pratique

```c
#include <stdio.h>

int main() {
    printf("Bonjour, je découvre le C !\n");
    printf("Je peux afficher plusieurs lignes.\n");
    return 0;
}
```

Exercices

1. Crée un programme qui affiche ton prénom.
2. Ajoute un second `printf` qui dit ton âge.
3. Mets un commentaire expliquant chaque ligne.
4. Supprime un `;` volontairement : que se passe-t-il ?
5. Supprime les `{}` et compile : que se passe-t-il ?

À retenir

- Un programme C commence toujours par `int main()`.
- Les instructions se terminent par `;`.
- Les `{}` regroupent un bloc d'instructions.
- Les commentaires servent à documenter ton code.
- Le C doit être compilé avant de s'exécuter.

Parfait 👍 Tu as raison : c'est un point essentiel à souligner pour bien marquer la différence. Voici la version mise à jour de l'introduction (modifiée dans la partie **compilé vs interprété** et **à retenir**).

Introduction au langage C

Objectifs

À la fin de ce chapitre, tu sauras :

- Situer le langage C dans l'histoire de l'informatique.
- Expliquer la différence entre un langage **compilé** et un langage **interprété**.
- Comprendre pourquoi on apprend encore le C aujourd'hui.
- Connaître les outils nécessaires pour programmer en C.

Le C en quelques mots

Le langage C a été créé dans les années 1970.

- Il a servi à écrire **le système Unix** et plus tard **Linux**.
- Beaucoup de logiciels systèmes (compilateurs, bases de données, microcontrôleurs) utilisent encore le C.
- Le C est un langage **rapide** et **proche de la machine** : il permet de comprendre comment l'ordinateur gère la mémoire et exécute les instructions.

Langages compilés et interprétés

Il existe deux grandes familles de langages :

- **Langages interprétés** (ex. Python, JavaScript)

    - Ton code est **lu ligne par ligne** par un interpréteur.
    - Tu dois avoir cet interpréteur installé pour lancer ton programme.
        - Exemple : `python moncode.py` → il faut Python installé.
    - Avantage : facile à tester rapidement.
    - Inconvénient : plus lent, dépend toujours de l'interpréteur.
- **Langages compilés** (ex. C, C++)
    - Ton code est **traduit en langage machine** avant d'être exécuté → création d'un fichier exécutable.
    - Une fois compilé, ton programme peut tourner **sans avoir besoin du compilateur**.
        - Exemple : `moncode.c` → compilation → `moncode.exe`, exécutable directement.
    - Avantage : programme **rapide et autonome**.
    - Inconvénient : il faut passer par l'étape de **compilation**.

💡 **Comparaison rapide :**

- Python → `python moncode.py` (l'interpréteur est indispensable).
- C → `gcc moncode.c -o moncode` puis `./moncode` (le programme compilé peut être exécuté seul).

Pourquoi apprendre le C

- Comprendre **comment fonctionne un ordinateur** en profondeur.
- Apprendre la rigueur d'un langage plus « strict » que Python ou JavaScript.
- Utile pour la **programmation système**, l'**embarqué** (microcontrôleurs, robots), et même pour certains moteurs de jeux vidéo.

Les outils à utiliser

Pour écrire et lancer un programme en C, tu as besoin de :

1. **Un éditeur de code / IDE** : Code::Blocks, Visual Studio Code, CLion…
2. **Un compilateur** : le plus courant est `gcc` (GNU Compiler Collection).
3. **Un terminal / console** pour compiler et exécuter ton programme.

👉 On fera un article spécifique pour t'expliquer **comment installer et utiliser ces outils**.

Exercices de réflexion

1. Classe ces langages dans la bonne catégorie : C, JavaScript, Python, C++.
2. Pourquoi dit-on qu'un programme compilé est plus rapide qu'un programme interprété ?

3. Imagine : tu veux programmer un petit jeu sur ton navigateur. Choisirais-tu plutôt C ou JavaScript ? Pourquoi ?
4. Explique pourquoi tu ne peux pas lancer directement `moncode.c` comme un fichier exécutable.

À retenir

- Le C est un langage **compilé**, contrairement à Python ou JavaScript qui sont **interprétés**.
- Un langage interprété → nécessite **toujours l'interpréteur installé** pour exécuter ton programme.
- Un langage compilé → produit un **exécutable autonome**, que tu peux lancer directement.
- Le C reste aujourd'hui incontournable pour comprendre **l'informatique au plus près de la machine**.

Veux-tu que je mette aussi un **schéma visuel simple** (pipeline "Code source → Interpréteur → Résultat" vs "Code source → Compilation → Exécutable → Résultat") pour renforcer la compréhension des élèves ?

Python - C: les différences de syntaxe

> *Pour les élèves ayant appris les bases de Python, découvrir le langage C peut être déstabilisant au début en raison de différences syntaxiques. Dans cet article, nous allons expliquer en détail ces différences sur des aspects clés comme les déclarations de variables, les blocs de code, les conditions, les boucles et les fonctions d'affichage. À la fin, vous comprendrez comment passer de Python à C avec des exemples concrets.*

Instructions (retour à la ligne vs `;`)

En python

En Python, les instructions sont séparées par un retour à la ligne:

```
annee_courante = 40
annee_naissance = 20

age = annee_courante - annee_naissance
```

En C, par contre, les instructions sont séparées par des '`;`' (points-virgules):

```
int annee_courante = 40;
int annee_naissance = 20;

int age = annee_courante - annee_naissance;
```

Les **retours à la ligne** ne sont pas obligatoires en C, même s'ils restent **FORTEMENT recommandés pour raisons de lisibilité**.

Blocs de code : indentations vs accolades `{}`

En Python

Les blocs de code sont définis par **l'indentation**. Aucune accolade n'est nécessaire.

Exemple :

```
if x > 0:
    print("x est positif")
    print("Fin du bloc")
```

En C

En C, les blocs de code sont délimités par des **accolades** `{}`. L'indentation est optionnelle mais recommandée pour améliorer la lisibilité.

Exemple équivalent en C :

```c
if (x > 0) {
    printf("x est positif");
    printf("Fin du bloc");
}
```

Conditions (`elif` vs `else if`)

En Python

En Python, lorsqu'une condition `if` est fausse, vous pouvez tester une autre condition en utilisant `elif`. C'est une syntaxe compacte qui permet d'enchaîner plusieurs tests dans un même bloc.

Exemple :

```python
age = 20

if age < 10:
    print("Enfant")
elif age < 18:
    print("Adolescent")
elif age < 30:
    print("Jeune adulte")
else:
    print("Adulte")
```

En C

En C, il n'existe pas de mot-clé unique équivalent à `elif`. Vous devez utiliser `else if` avec un espace. Chaque bloc `else if` est suivi de la condition entre parenthèses et utilise des accolades `{}` pour le code.

Exemple équivalent en C :

```c
int age = 20;

if (age < 10) {
    printf("Enfant");
} else if (age < 18) {
    printf("Adolescent");
} else if (age < 30) {
    printf("Jeune adulte");
} else {
    printf("Adulte");
}
```

Points à noter

- En Python, `elif` est plus compact, alors que le C utilise deux mots-clés séparés (`else if`).
- En C, les parenthèses autour des conditions sont obligatoires.
- Les accolades `{}` délimitent les blocs de code en C, tandis que l'indentation est suffisante en Python.

Afficher des informations ("à l'écran")

Les micro-contrôlleurs n'ont généralement pas d'écran. L'affichage des informations (en mode développement) se fait généralement sur l'ordinateur hôte auquel le micro-contrôlleur est connecté.

En Python

En Python, la fonction `print()` est utilisée pour afficher des messages ou des valeurs dans la console. Vous pouvez facilement afficher plusieurs valeurs en les séparant par des virgules.

Exemple :

```c
temperature = 22.5
print("La température est :", temperature)
```

En C - La fonction `printf`

La fonction `printf` (pour **print formatted**) sert à **afficher du texte ou des valeurs** à l'écran. Elle fait partie de la bibliothèque `stdio.h`, qu'il faut donc inclure au début du programme :

```c
#include <stdio.h>
```

Sa syntaxe de base est :

```c
printf("texte à afficher");
```

Tu peux afficher plusieurs types de données en utilisant des **spécificateurs de format** :

- `%d` → un entier
- `%f` → un nombre décimal (float)
- `%c` → un caractère
- `%s` → une chaîne de caractères

Exemple :

```
int age = 16;
printf("J'ai %d ans.\n", age);
```

Le `\n` à la fin du texte sert à **aller à la ligne suivante**.

Déclaration des variables

En Python

Pas besoin de déclarer explicitement le type de la variable : Python le déduit automatiquement lors de l'assignation.

Exemple :

```
x = 10          # Entier
y = 22.5        # Flottant
message = "OK" # Chaîne de caractères
```

En C

En C, il faut **toujours déclarer le type** de chaque variable avant de l'utiliser. Cette règle permet au compilateur d'optimiser la mémoire.

Exemple équivalent en C :

```
int x = 10;          // Entier
float y = 22.5;      // Flottant
char message[] = "OK"; // Chaîne de caractères
```

Différence clé

En Python, le typage est dynamique et flexible. En C, le typage est strict et doit être spécifié à l'avance.

Opérateurs logiques : `and`, `or` vs `&&`, `||`

En Python

Les opérateurs logiques en Python sont écrits en toutes lettres : `and` pour ET, `or` pour OU.

Exemple :

```
if x > 0 and y < 10:
    print("x est positif ET y est inférieur à 10")

if x > 0 or y < 10:
    print("x est positif OU y est inférieur à 10")
```

En C

En C, les opérateurs logiques sont symboliques :

- `&&` pour ET
- `||` pour OU

Exemple équivalent en C :

```
if (x > 0 && y < 10) {
    printf("x est positif ET y est inférieur à 10");
}

if (x > 0 || y < 10) {
    printf("x est positif OU y est inférieur à 10");
}
```

Différence clé

Python utilise des mots-clés (`and` , `or`), alors que C utilise des symboles (`&&` , `||`).

Boucle `while`

En Python

Une boucle `while` s'écrit simplement et utilise l'indentation pour définir le bloc de code.

Exemple :

```
x = 0
while x < 5:
    print(x)
    x += 1
```

En C

En C, la boucle `while` est similaire, mais les blocs de code sont définis par des accolades `{}`.

Exemple équivalent en C :

```
int x = 0;
while (x < 5) {
    printf("%d", x);
    x += 1;
}
```

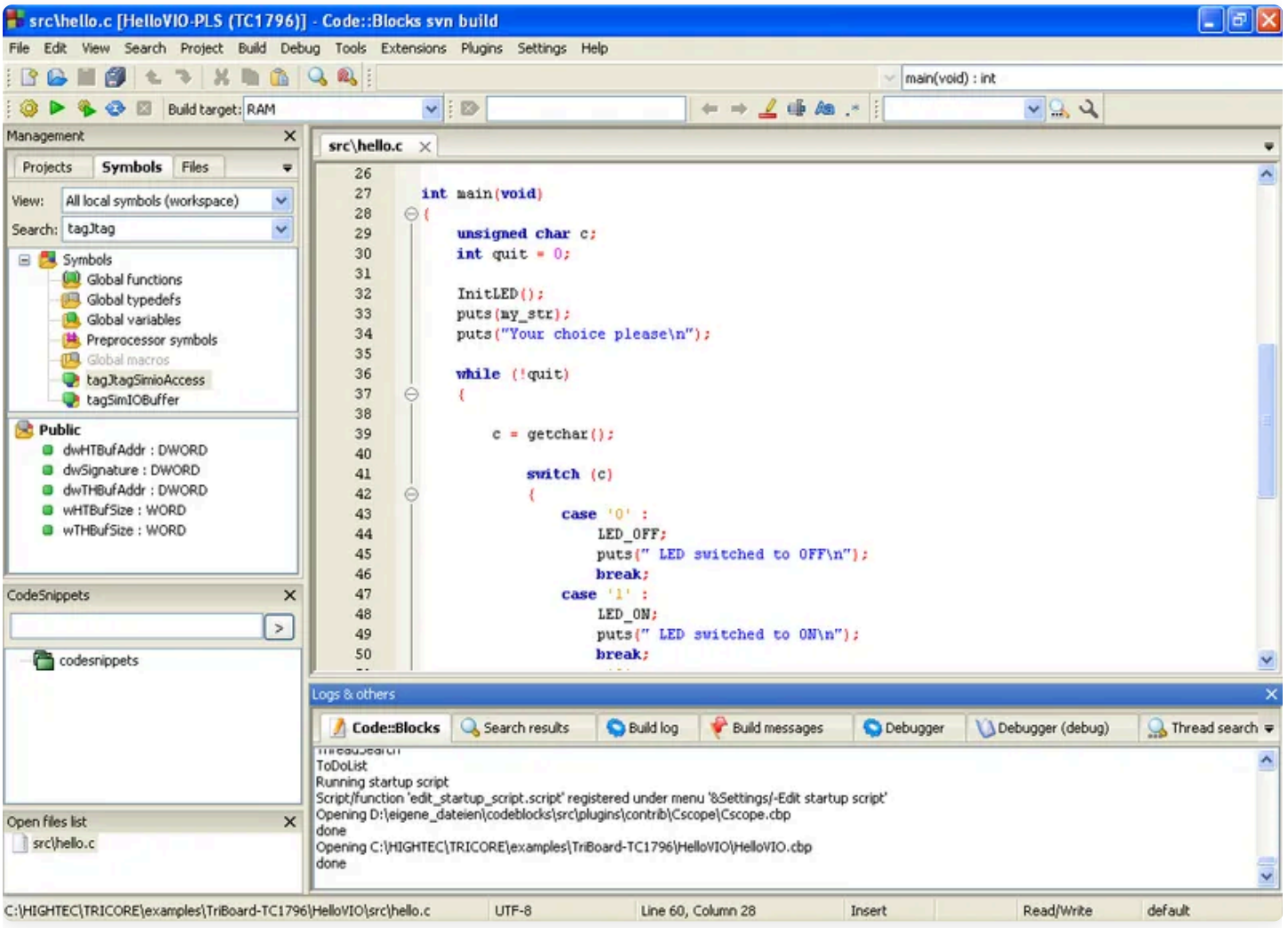
Conclusion

Les différences entre Python et le C se situent principalement au niveau de la syntaxe. Le C est plus strict avec ses exigences sur les types, l'utilisation des accolades `{}` et les points-virgules `;`, tandis que Python privilégie la lisibilité et la flexibilité. Cependant, une fois ces différences comprises, vous pourrez facilement adapter vos compétences en Python pour écrire des programmes robustes en C. La pratique régulière et des exemples concrets, comme ceux présentés ici, vous aideront à maîtriser ces deux langages.

Écrire du C - Code::Blocks IDE

> *Code::Block est un environnement de développement C/C++ gratuit et open source. Il inclut le compilateur gratuit MinGW, compatible avec les outils GCC ('GNU Compiler Collection', généralement rencontrés sous Linux...)*

Installer et utiliser Code::Blocks



Objectifs

À la fin de ce chapitre, tu sauras :

- Installer Code::Blocks avec le compilateur inclus.
- Créer ton premier projet en C.
- Compiler et exécuter un programme.

- Comprendre l'organisation de l'interface.
- Éviter les erreurs fréquentes au démarrage.

Pourquoi Code::Blocks ?

Code::Blocks est un **environnement de développement (IDE)** spécialement conçu pour les langages compilés comme C et C++.

- Il regroupe tout ce dont tu as besoin : **éditeur de code**, **compilateur**, **outil pour exécuter tes programmes**.
- Avantage : tu n'as pas besoin de configurer séparément un compilateur.

Installation

💥 Pense bien à télécharger la version **avec le compilateur MingW**.

1. Va sur le site officiel : https://www.codeblocks.org/downloads/binaries/.
2. Choisis la version **Code::Blocks with MinGW** (c'est celle qui contient déjà le compilateur `gcc`): exemple: `codeblocks-25.03mingw-setup.exe`.
   - ⚠️ Ne télécharge pas la version "sans MinGW", sinon il te manquera le compilateur.
3. Lance l'installation (clic sur "Suivant" plusieurs fois).
4. Laisse les options par défaut.
5. À la fin, ouvre Code::Blocks pour vérifier que tout fonctionne.

Tu peux télécharger la version installable, auquel tu devras demander un accès admin au moment de l'installation, ou la version "nosetup" (portable) auquel cas tu devras créer les raccourcis toi-même.

Première utilisation

Quand tu ouvres Code::Blocks :

- Barre de menus en haut (Fichier, Éditer, Projet…).
- Zone principale pour écrire ton code.
- Panneau de messages en bas (erreurs, avertissements).
- Boutons rapides (nouveau, ouvrir, compiler, exécuter).

Créer ton premier projet

1. Clique sur **File > New > Project**.

2. Choisis **Console Application** → **Next**.
3. Sélectionne **C language** → **Next**.
4. Donne un **nom à ton projet** (ex: `HelloWorld`).
5. Choisis un dossier où sauvegarder.
6. Clique sur **Finish**.

Code::Blocks génère automatiquement un fichier `main.c` avec un programme exemple.

Compiler et exécuter

- Pour **compiler** ton programme : clique sur le bouton en forme de **roue dentée**.
- Pour **lancer** ton programme : clique sur le bouton **triangle vert**.
- Pour faire les deux en même temps : clique sur **Build & Run (F9)**.

👉 Si tout va bien, une fenêtre noire (console) s'ouvre et affiche :

```
Hello world!
```

Erreurs fréquentes

- **Si Code::Blocks ne trouve pas le compilateur** :
  - Vérifie que tu as bien téléchargé la version *with MinGW*.
- **Fenêtre console qui se ferme trop vite** :
  - Ajoute une ligne `getchar();` avant `return 0;` dans `main.c`.

Exercices

1. Installe Code::Blocks sur ton ordinateur personnel.
2. Crée un projet console nommé `Test`.
3. Modifie le fichier `main.c` pour afficher ton prénom.
4. Ajoute une deuxième ligne qui affiche ton âge.
5. Compile et exécute ton programme.

À retenir

- Télécharge toujours la version **Code::Blocks with MinGW**.
- Un projet console est le type de projet à utiliser pour apprendre le C.

- La compilation est indispensable avant l'exécution.
- Les erreurs affichées en bas de l'écran sont **précieuses** pour corriger ton code.

Notes pour l'enseignant

Objectifs pédagogiques

- Permettre à chaque élève d'avoir un environnement de développement fonctionnel.
- Introduire la notion de **projet** et d'organisation des fichiers.
- Dédramatiser les premiers messages d'erreurs de compilation.

Points-clés à insister

- Toujours choisir la version avec MinGW (évite 90 % des problèmes débutants).
- Insister sur le rôle du bouton **Build & Run** (F9).
- Les erreurs de compilation sont normales, il faut les lire attentivement.

Bonnes pratiques

- Demander aux élèves de **nommer clairement** leurs projets (pas `projet1`, mais `HelloWorld`, `Calculatrice`…).
- Leur faire tester volontairement une erreur (oublier un `;`) pour voir le message affiché.
- Montrer que Code::Blocks est un **outil de travail**, pas seulement une "boîte magique".

C - Les variables

> *Dans la programmation C, les variables permettent de stocker et manipuler des données. Dans cet article, nous allons explorer les principaux types de variables en C, leur déclaration, assignation, et usage, avec des exemples pratiques*

Qu'est-ce qu'une variable ?

Une **variable** est une zone de mémoire dans laquelle vous pouvez stocker une valeur qui peut changer au cours de l'exécution de votre programme. Les variables ont trois caractéristiques principales :

- **Nom** : l'identifiant de la variable (exemple : `vitesseMoteur`).
- **Type** : le type de données qu'elle peut contenir (exemple : entier, flottant, etc.).
- **Valeur** : la donnée actuelle stockée dans la variable.

Déclaration et assignation de variables

1. Déclaration

Déclarer une variable signifie informer le programme que vous souhaitez réserver de la mémoire pour cette variable, en précisant **son type et son nom**.

Syntaxe :

```
type nomVariable;
```

2. Assignation

Attribuer une valeur à une variable se fait grâce à l'opérateur `=`.

Syntaxe :

```
nomVariable = valeur;
```

Vous pouvez aussi combiner la déclaration et l'assignation en une seule étape.

Syntaxe combinée :

```
type nomVariable = valeur;
```

C - Les types de variables

> *Dans la programmation C, les variables permettent de stocker et manipuler des données. Dans cet article, nous allons explorer les principaux types de variables en C avec des exemples pratiques.*

Types de variables en C

C est un langage à **typage statique** (*statically typed*): les types des variables sont **contrôlés au moment de la compilation**.

Une fois la variable déclarée avec un type, elle ne peut plus changer de type par la suite.

Voici les principaux types de variables utilisés dans la programmation en C:

▸ Les entiers (`int` , `unsigned int`)

Les entiers permettent de stocker des nombres entiers, positifs ou négatifs.

Les **int** stockent des valeurs positives et/ou négatives (ils utilisent un bit de signe).

Les **unsigned int** stockent des valeurs positives uniquement.

▸ Les flottants (`float` , `double`)

Les flottants sont utilisés pour stocker des nombres décimaux. Ils sont utiles pour les calculs de précision.

- **float** : précision jusqu'à 6-7 chiffres après la virgule.
- **double** : souvent identique à `float` (pas de différence).

Exemple : Calcul de la distance parcourue par un robot.

```c
float distanceParcourue = 12.34; // En mètres
double angleRotation = 45.5; // En degrés
```

▸ Les caractères (`char`)

Les caractères permettent de stocker une lettre ou un symbole ASCII. Ils sont souvent utilisés pour gérer des messages ou des commandes.

Exemple : Communication avec un robot via Bluetooth.

```c
char commandeRecue = 'A'; // Commande reçue : avancer
```

▸ Les chaînes de caractères (`char[]` ou `String`)

Les chaînes de caractères sont utilisées pour manipuler des mots ou des messages.

- **char[]** : tableau de caractères, plus optimisé pour les microcontrôleurs.

- **String** : classe de manipulation de chaînes, plus facile à utiliser mais consomme plus de mémoire.

Exemple : Affichage d'un message sur un écran LCD.

```
char messageLCD[] = "Robot prêt"; // Message statique
```

Les booléens (`bool`)

Les booléens stockent des valeurs `true` (vrai) ou `false` (faux). Ils sont utiles pour gérer des états.

Exemple : Vérification de la détection d'un obstacle.

```
bool obstacleDetecte = false; // Initialisation
```

Bonnes pratiques pour utiliser les variables

1. **Noms explicites** : Donnez des noms significatifs à vos variables pour comprendre leur rôle.
   - Mauvais exemple : `int a;`
   - Bon exemple : `int vitesseMoteur;`
2. **Initialisation** : Assurez-vous que vos variables sont initialisées avant leur utilisation pour éviter des comportements imprévisibles.
3. **Limitation de la portée** : Déclarez vos variables au niveau approprié (locale ou globale) pour éviter les conflits et réduire la consommation de mémoire.
4. **Évitez les types coûteux** : Préférez des types simples (comme `char[]`) plutôt que `String` si la mémoire est limitée.

Variables globales et locales

Variables globales

Déclarées en dehors de toute fonction, les variables globales sont accessibles dans tout le programme.

Exemple : Stocker la vitesse du robot.

```
int vitesseGlobale = 100; // Accès depuis n'importe quelle fonction
```

Variables locales

Déclarées à l'intérieur d'une fonction, elles ne sont accessibles que dans cette fonction.

Exemple : Utiliser une variable temporaire pour un calcul.

```
void avancer() {
    int vitesseLocale = 120; // Variable spécifique à cette fonction
    analogWrite(3, vitesseLocale);
}
```

Conclusion

Les variables sont au cœur de vos projets de programmation. Bien choisir leur type et leur portée, les initialiser correctement, et les utiliser de manière structurée est essentiel pour garantir la fiabilité et la clarté de votre code. Avec ces bases et les exemples fournis, vous êtes prêt à gérer efficacement les données de vos programmes.

C - Les nombres entiers (`int`)

> *Les **types entiers** en C permettent de représenter des **nombres sans virgule**, **positifs ou négatifs**. Le C offre des versions **signées** et **non signées** (**`unsigned`**) pour contrôler la présence ou non de nombres négatifs. Bien comprendre ces différences est essentiel pour écrire des programmes efficaces et fiables.*

Les entiers (`int`, `unsigned int`)

Les entiers permettent de stocker des nombres entiers (sans virgule), positifs ou négatifs.

Les **int** stockent des valeurs positives et/ou négatives (ils utilisent un bit de signe).

Les **unsigned int** stockent des valeurs positives uniquement

La taille d'un `int` en C dépend de l'architecture de la machine, mais il est généralement de **4 octets sur les systèmes modernes 32 bits et 64 bits**. La taille peut être de 2 octets sur des systèmes 16 bits plus anciens ou embarqués.

TYPE	TAILLE (16 BITS)	VALEUR MINIMALE (16 BITS)	VALEUR MAXIMALE (16 BITS)	TAILLE (32/64 BITS)	VALEUR MINIMALE (32/64 BITS)	VALEUR MAXIMALE (32/64 BITS)
`int`	2 octets (16 bits)	−32 768	+32 767	4 octets (32 bits)	−2 147 483 648	+2 147 483 647
`unsigned int`	2 octets	0	65 535	4 octets	0	4 294 967 295

À retenir

- La taille d'un type dépend du **compilateur** et de la **plateforme**.
- Un `unsigned` ne peut pas représenter de nombres négatifs.
- Utilise `sizeof(type)` dans ton programme pour vérifier la taille réelle :

```
printf("int = %zu octets\n", sizeof(int));
```

Exemples

```
int temperature = - 10;
unsigned int vitesseMoteur = 100; // Vitesse initiale du moteur
```

C - Les nombres à virgule (`float` et `double`)

> *Les types à virgule (ou types flottants) servent à représenter des nombres réels, c'est-à-dire des valeurs qui contiennent une **partie décimale**. En C, on utilise principalement trois types: **`float`**, **`double`** et **`long double`**. Ils se distinguent par leur précision (le nombre de chiffres après la virgule) et leur taille en mémoire.*

Le type `float` est le plus léger, `double` le plus courant, et `long double` le plus précis.

Les flottants (`float`, `double`)

Les flottants sont utilisés pour stocker des nombres décimaux. Ils sont utiles pour les calculs de précision.

La principale différence entre `float` et `double` en C est la **précision** et la **taille en mémoire**.

Un type `float` utilise **4 octets** (32 bits) et offre une précision d'environ **6-7 chiffres** après la virgule.

Un type `double` offre une double précision et utilise **8 octets** (64 bits) pour une **précision plus élevée d'environ 15 chiffres** après la virgule.

Un type `long double` offre une précision encore meilleure (jusqu'à 33 chiffres après la virgule, en fonction du système).

Le choix du type dépend des besoins en précision et des contraintes de mémoire de votre programme.

Exemple : Calcul de la distance parcourue par un robot.

```c
float distanceParcourue = 12.34; // En mètres
double pi = 3.14159265359;
```

C - Les caractères (`char`)

> *Le type char en C sert à représenter un caractère unique. Lorsque l'on veut stocker plusieurs caractères à la suite, comme un mot ou une phrase, on utilise un tableau de caractères, noté char[].*

▸ Les caractères (`char`)

Les caractères permettent de stocker une lettre ou un symbole ASCII. Ils sont souvent utilisés pour gérer des messages ou des commandes.

Exemple : Communication avec un robot via Bluetooth.

```
char commandeRecue = 'A'; // Commande reçue : avancer
```

En C, on utilise les **apostrophes** (*single quotes*) pour les cartactères.

En réalité, un `char` contient le code ASCII du caractère:

```
    char code = 'A';
    printf("%c", code); // --> Affiche A
    printf("%d", code); // --> Affiche 65
```

▸ Les chaînes de caractères (`char[]`)

Les chaînes de caractères sont utilisées pour manipuler des mots ou des messages. Contrairement à d'autres langages comme Python, C ne possède pas de type "*string*". Les chaînes de caractères sont stockées sous forme de tableau de caractères (`char[]`).

Il faut utiliser des guillemets (*double quotes*) pour définir uns *string*:

```
char prenom[] = "Marinette";
char messageLCD[] = "Robot prêt"; // Message statique
```

⚠️ Important Les chaînes de caractères sont TOUJOURS terminées par le caractère `\0`. C l'ajoute automatiquement quand tu déclares une chaîne avec des `""`. Cela aura de l'importance plus tard quand tu réserveras toi-même l'espace mémoire pour y stocker ta chaîne: il faut réserver un espace suffisament grand que pour y stocker aussi le `\0`.

D'ailleurs cela peut se voir en affichant la taille d'une chaïne:

```
char chaine[] = "azerty";
printf("%d", sizeof(chaine)); // Affiche 7
```

C - Assigner une valeur à des strings

> *Contrairement à Python ou JavaScript, le langage C ne possède pas un type "string" intégré. Les textes sont gérés sous forme de tableaux de caractères, ce qui change entièrement la manière de les créer, de les modifier et de les copier. Cet article t'aide à comprendre cette logique particulière.*

En C, un tableau de char (`char nom[30];`) n'est pas une variable assignable

Un tableau **ne peut pas être affecté** avec une simple instruction d'assignation.

Donc ceci est **illégal** :

```
char nom[30];
nom = "Alice";    // ❌ interdit
```

Pourquoi ? Parce qu'un tableau **n'est pas une variable "classique"**, c'est un bloc mémoire fixe. On ne peut pas "remplacer" un tableau : on peut seulement **écrire à l'intérieur** case par case.

Les seuls moments où un tableau peut recevoir une chaîne littérale sont :

👉 **lors de la déclaration + initialisation**

```
char nom[30] = "Alice";   // ✔ OK (initialisation, pas assignation)
```

👉 **ou en utilisant une fonction de copie**

```
strcpy(nom, "Alice");     // ✔ OK
```

Donc : 💡 *Toutes les affectations de chaînes dans un tableau déjà existant nécessitent une fonction de copie.*

Que se passe-t-il avec `strcpy()` ?

`strcpy(destination, source);`

Elle copie **caractère par caractère** :

- de `source` ,
- vers `destination` ,
- jusqu'au `\0` final.

Donc :

```
strcpy(nom, "Alice");
```

Écrit réellement ceci en mémoire :

```
nom[0] = 'A'
nom[1] = 'l'
nom[2] = 'i'
nom[3] = 'c'
nom[4] = 'e'
nom[5] = '\0'
```

Exception importante : les *pointeurs* de type `char*`

Si tu déclares ceci :

```
char *nom = "Alice";
```

Ici, tu n'as **pas** un tableau. Tu as un **pointeur** qui pointe vers une chaîne littérale *stockée en mémoire en lecture seule*.

Donc pour un `char*`, ceci est permis :

```
nom = "Bob";   // ✔ valide (on change l'adresse stockée dans le pointeur)
```

Mais **tu ne peux pas modifier le contenu** de la chaîne littérale !

```
nom[0] = 'C';   // ⚠ comportement indéfini → crash probable
```

Donc :

TYPE	PEUT-ON FAIRE `NOM = "TEXTE"` ?	PEUT-ON MODIFIER LE CONTENU ?
`char nom[30]` (tableau)	❌ Non	✔ Oui
`char *nom` (pointeur)	✔ Oui	❌ Non (dangereux)

Conclusion

▸ **Si tu utilises un tableau de char pour stocker une chaîne :**

🎯 **Obligatoire** : utiliser une fonction de copie (`strcpy` , `strncpy` , `fgets` , etc.)

Parce qu'un tableau ne peut pas être affecté avec `=` .

- ▸ **Si tu utilises un pointeur `char*` vers un littéral :**
- Tu peux faire `nom = "Alice";`
- Mais tu ne peux **pas modifier** les caractères du littéral.

Alors, quand utiliser `strcpy` ?

💡 **Toujours** quand tu veux mettre du texte dans un tableau de char :

```
char nom[30];
strcpy(nom, "Alice");      // ✔ bon
```

Ou quand tu veux copier une variable string dans une autre.

Bonus — Recommandation moderne

Pour éviter les dépassements de mémoire :

```
strncpy(nom, "Alice", sizeof(nom));
nom[sizeof(nom) - 1] = '\0';
```

Ou mieux :

```
fgets(nom, sizeof(nom), stdin);
```

C - Les booléean (`bool`)

> *Les booléens stockent des valeurs `true` (vrai) ou `false` (faux).*

▸ Les booléens (`bool`)

Les booléens stockent des valeurs `true` (vrai) ou `false` (faux). Ils sont utiles pour gérer des états.

Exemple : Vérification de la détection d'un obstacle.

```
bool obstacleDetecte = false; // Initialisation
```

💪 C - Les variables (Exercices)

> *Quelques exercices pour travailler les variables*

Voici une série de **10 exercices progressifs** pour apprendre à manipuler les **variables en langage C**, spécialement conçus pour des élèves débutants 👇

Exercices – Les variables en C

Objectifs

Ces exercices te permettront de :

- Déclarer et utiliser des variables de différents types (`int` , `float` , `char` , `double`).
- Comprendre la notion de valeur, d'affectation et de type.
- Faire des calculs simples et afficher les résultats avec `printf` .

Exercice 1 – Afficher une variable

Déclare une variable entière `age` , donne-lui la valeur `16` , puis affiche :

```
J'ai 16 ans.
```

Exercice 2 – Changer la valeur d'une variable

Déclare une variable `nombre` , mets-lui la valeur `5` , affiche-la, puis change sa valeur en `10` et affiche-la à nouveau.

Résultat attendu :

```
Valeur initiale : 5
Nouvelle valeur : 10
```

Exercice 3 – Somme de deux nombres

Déclare deux entiers `a` et `b`, donne-leur les valeurs `7` et `3`, calcule leur somme dans une troisième variable `somme`, puis affiche :

```
La somme vaut 10
```

Exercice 4 – Calcul de moyenne

Déclare trois notes (ex: `note1`, `note2`, `note3`) de type `float`. Calcule la moyenne et affiche :

```
Moyenne = 14.33
```

(Utilise `%f` dans `printf`.)

Exercice 5 – Aire d'un rectangle

Demande à l'utilisateur la **longueur** et la **largeur**, puis calcule et affiche l'aire.

```
Entre la longueur : 5
Entre la largeur : 3
Aire = 15
```

Exercice 6 – Division réelle

Déclare deux `float` : `a = 7`, `b = 2`. Affiche le résultat de la division `a / b`. ⚠️ Si tu utilises des `int`, le résultat sera arrondi ! Exemple attendu :

```
7 / 2 = 3.50
```

Exercice 7 – Calcul avec plusieurs types

Déclare :

```c
int entier = 5;
float reel = 2.5;
```

Crée une variable `resultat` (float) et calcule `entier * reel`. Affiche le résultat.

Exercice 8 – Caractère et code ASCII

Déclare une variable `char lettre = 'A';` Affiche :

```
La lettre est A et son code ASCII est 65
```

(Utilise `%c` pour la lettre et `%d` pour son code.)

Exercice 9 – Échanger deux valeurs

Déclare deux entiers `x = 3` et `y = 8`. Échange leurs valeurs **en utilisant une troisième variable temporaire**, puis affiche :

```
Avant échange : x=3, y=8
Après échange : x=8, y=3
```

Exercice 10 – Conversion d'euros en dollars

Demande à l'utilisateur un montant en euros (float). Utilise un **taux de conversion fixe**, par exemple `1 euro = 1.1 dollars`. Affiche le résultat :

```
Montant en euros : 25
Montant en dollars : 27.50
```

Bonus – Pour aller plus loin

Teste ce qui se passe si :

- tu oublies d'initialiser une variable avant de l'utiliser,
- tu utilises le mauvais type (ex: `%d` pour un `float`),
- tu changes la valeur d'une variable après un `printf`.

C - Opérations mathématiques

> *En programmation, les opérations mathématiques permettent de faire des calculs avec des variables : additionner, soustraire, multiplier, diviser, ou encore obtenir le reste d'une division. Le langage C fonctionne comme les mathématiques classiques, mais il faut faire attention aux types de données (entiers, réels) et à la façon dont la division est calculée.*

Parfait 👍 Voici un **article de cours complet** sur les **opérations mathématiques en langage C**, rédigé selon ta méthodologie (enseignement explicite, format pour élèves + notes enseignant).

Les opérations mathématiques en C

Objectifs

À la fin de ce chapitre, tu sauras :

- Utiliser les opérateurs arithmétiques de base (`+` , `-` , `*` , `/` , `%`).
- Comprendre la différence entre division entière et division réelle.
- Combiner plusieurs opérations dans une même expression.
- Utiliser les parenthèses pour contrôler l'ordre des calculs.
- Afficher les résultats avec `printf` .

Les opérateurs de base

En C, les opérations mathématiques se font à l'aide d'**opérateurs** :

OPÉRATEUR	SIGNIFICATION	EXEMPLE	RÉSULTAT
`+`	addition	`5 + 2`	`7`
`-`	soustraction	`5 - 2`	`3`
`*`	multiplication	`5 * 2`	`10`
`/`	division	`5 / 2`	`2` (division entière)
`%`	reste de la division	`5 % 2`	`1`

Exemple simple

```c
#include <stdio.h>

int main() {
    int a = 10, b = 3;

    printf("Addition : %d\n", a + b);
    printf("Soustraction : %d\n", a - b);
    printf("Multiplication : %d\n", a * b);
    printf("Division entière : %d\n", a / b);
    printf("Reste (modulo) : %d\n", a % b);

    return 0;
}
```

▸ Résultat

```
Addition : 13
Soustraction : 7
Multiplication : 30
Division entière : 3
Reste (modulo) : 1
```

Division entière vs réelle

⚠️ En C, si tu divises **deux entiers**, le résultat est **un entier** :

```c
int a = 7, b = 2;
printf("%d", a / b);  // Affiche 3 (et pas 3.5)
```

Pour obtenir une **division réelle**, il faut utiliser des `float` ou `double` :

```c
float x = 7, y = 2;
printf("%.2f", x / y);  // Affiche 3.50
```

💡 `%.2f` affiche le résultat **arrondi** à **2 chiffres après la virgule**.

L'ordre des opérations

Comme en mathématiques, le C suit les priorités habituelles :

1. Parenthèses `( )`
2. Multiplication et division `*`, `/`, `%`
3. Addition et soustraction `+`, `-`

Exemple :

```c
int resultat = 5 + 2 * 3;      // = 11
int resultat2 = (5 + 2) * 3;   // = 21
```

Opérations combinées

Tu peux évidemment faire plusieurs calculs dans la même expression :

```c
int a = 4, b = 2, c = 3;
int r = (a + b) * c - 1;
printf("Résultat : %d\n", r);
```

Affectation et raccourcis utiles

Il existe des **formes abrégées** pour modifier la valeur d'une variable :

FORME CLASSIQUE	RACCOURCI	SIGNIFICATION
`x = x + 3;`	`x += 3;`	ajoute 3 à x
`x = x - 4;`	`x -= 4;`	retire 4 à x
`x = x * 2;`	`x *= 2;`	multiplie x par 2
`x = x / 3;`	`x /= 3;`	divise x par 3

Incrémentation et décrémentation (`i++`, `++i`, `i--`, `--i`)

En programmation, il est très courant de **modifier la valeur d'une variable numérique** à chaque étape d'un calcul ou d'une boucle. Au lieu d'écrire des formules longues comme `i = i + 1`, le langage C propose des **raccourcis mathématiques** très utilisés : les opérateurs d'**incrémentation** et de **décrémentation**.

Incrémenter et décrémenter une variable

- `i++` → ajoute **1** à la variable `i`
- `++i` → ajoute aussi **1** à la variable `i`

- `i--` → retire **1** à la variable `i`
- `--i` → retire aussi **1** à la variable `i`

Exemples :

```
int i = 5;
i++;  // i devient 6
i--;  // i redevient 5
```

Ces opérateurs sont donc une forme abrégée d'opérations arithmétiques très fréquentes :

- `i++` équivaut à `i = i + 1`
- `i--` équivaut à `i = i - 1`

▸ Différence entre `i++` et `++i`

Ces deux écritures **ont le même effet final** (augmenter `i` de 1), mais **pas au même moment** dans le calcul.

OPÉRATEUR	ÉTAPE DU CALCUL	EXEMPLE	VALEUR UTILISÉE
`i++`	**post-incrémentation** : on utilise `i`, puis on ajoute 1	`x = i++;`	`x` reçoit l'ancienne valeur de `i`
`++i`	**pré-incrémentation** : on ajoute 1, puis on utilise `i`	`x = ++i;`	`x` reçoit la nouvelle valeur de `i`

Exemple concret :

```
int i = 5;
int a = i++;  // a = 5, puis i = 6
int b = ++i;  // i = 7, puis b = 7
```

▸ Même chose pour la décrémentation

De la même façon :

```
int i = 3;
int a = i--;  // a = 3, puis i = 2
int b = --i;  // i = 1, puis b = 1
```

▸ À retenir

- `i++` ou `++i` → ajoutent 1 à `i`
- `i--` ou `--i` → retirent 1 à `i`
- La différence entre préfixe (`++i`) et postfixe (`i++`) n'a d'importance que **dans une expression mathématique**.

- Ces opérateurs sont très utiles pour **faire évoluer une variable** dans une boucle ou un calcul.

Exercices

1. Calcule la somme, la différence et le produit de 8 et 4.
2. Calcule le quotient et le reste de la division de 17 par 5.
3. Affiche le résultat de `(3 + 2) * 5`.
4. Déclare deux `float` et affiche leur moyenne avec 2 décimales.
5. Si `x = 10`, affiche le résultat après `x += 5`, puis après `x--`.

À retenir

- Les opérateurs arithmétiques sont : `+`, `-`, `*`, `/`, `%`.
- Une division entre deux entiers donne un **résultat entier**.
- Utilise des `float` ou `double` pour obtenir des nombres à virgule.
- Les **parenthèses** permettent de choisir l'ordre du calcul.
- Les opérateurs `+=`, `-=`, `++`, `--` sont des raccourcis pratiques.

C - Les noms de variables

> *En C, chaque variable doit porter un nom clair et précis qui permet de comprendre facilement ce qu'elle représente. Le choix des noms est essentiel pour écrire un code lisible, cohérent et professionnel. Respecter les règles de syntaxe et adopter de bonnes conventions de nommage facilite la maintenance du programme et la collaboration entre développeurs.*

Objectifs

À la fin de ce chapitre, tu sauras :

- Comprendre les **règles de nommage** des variables en langage C.
- Reconnaître les **erreurs à éviter** dans les noms de variables.
- Appliquer les **bonnes conventions d'écriture** pour rendre ton code clair et lisible.

Rappel : qu'est-ce qu'une variable ?

Une **variable** est un espace de mémoire qui contient une valeur d'un type donné. Elle a :

- un **nom** (identifiant),
- un **type** (ex : `int`, `float`, `char`),
- et une **valeur**.

Règles de base pour nommer une variable

Le langage C impose certaines règles strictes :

✅ **Un nom de variable doit :**

- Commencer par **une lettre** (majuscule ou minuscule) ou un **underscore** `_` ;
- Contenir uniquement des **lettres, chiffres ou** `_` ;
- Ne pas dépasser 31 caractères (pour rester lisible) ;
- Être **unique** dans un même bloc de code.

❌ **Un nom de variable ne peut pas :**

- Commencer par un chiffre ;
- Contenir des espaces ou des caractères spéciaux (`-`, `?`, `!`, etc.) ;
- Utiliser un **mot réservé du C** (ex : `int`, `return`, `if`, `while`, …).

Exemples :

```c
int age;          // ✅ correct
float moyenne_1;  // ✅ correct
int 2notes;       // ❌ incorrect (commence par un chiffre)
char prénom;      // ❌ incorrect (accent interdit)
int if;           // ❌ incorrect (mot réservé)
```

Le C est ***case-sensitive*** (sensible à la casse): il fait la **différence entre majuscules et minuscules**. C'est le cas aussi pour les noms de variables. => `prix` et `Prix` sont deux noms de variables différents.

Sens des noms

Un bon nom de variable doit **décrire ce qu'elle contient**. C'est essentiel pour que ton code soit **compréhensible** pour toi et pour les autres.

Exemples :

```c
float temperature;    // clair
float t;              // trop court : on ne sait pas ce que ça signifie
```

Conventions d'écriture

Même si le compilateur ne les impose pas, certaines **conventions** rendent ton code plus lisible.

🟢 Pour les variables et fonctions :

➡️ **utilise le style "camelCase" ou "snake_case"**

- `camelCase` : la première lettre minuscule, les mots suivants en majuscule. → `noteMoyenne`, `nombreEtudiants`
- `snake_case` : les mots séparés par un underscore `_`. → `note_moyenne`, `nombre_etudiants`

Les deux styles sont corrects, mais il est important d'en choisir **un seul** et de s'y tenir dans tout ton projet => l'important c'est la **consistance/cohérence/uniformité/régularité**.

🟡 Pour les constantes :

➡️ écris-les en **majuscules** avec des underscores :

```c
#define TAUX_TVA 0.21
const int VITESSE_LUMIERE = 299792458;
```

🔵 Pour les noms de fonctions :

➡️ commence par un **verbe**, pour indiquer une action :

```
calculerMoyenne();
afficherResultat();
```

Bonnes pratiques générales

- Choisis des noms **clairs et précis** : `note_finale` est mieux que `n` ou `x`.
- Reste **cohérent** : n'alterne pas entre `camelCase` et `snake_case`.
- N'utilise pas de majuscules au hasard : `Age` et `age` sont **deux variables différentes** en C.
- Évite les abréviations obscures : `nb_eleve` vaut mieux que `nbe` ou `ne`.
- Sois attentif à l'**orthographe** : `moyenne`, pas `moyene`.

Exemple de bonnes pratiques

```c
#include <stdio.h>

int main() {
    int nombre_etudiants = 25;
    float moyenne_generale = 13.8;
    char premiere_lettre = 'M';

    printf("Il y a %d étudiants.\n", nombre_etudiants);
    printf("La moyenne générale est de %.1f.\n", moyenne_generale);
    printf("La première lettre est %c.\n", premiere_lettre);

    return 0;
}
```

Ce code est **facile à lire** et les noms sont **cohérents** et **significatifs**.

À retenir

- Un nom de variable doit **commencer par une lettre**, ne pas contenir de caractères spéciaux ni de mots réservés.
- Un nom **clair** facilite la lecture et la compréhension du code.
- Choisis une **convention d'écriture** (`camelCase` ou `snake_case`) et respecte-la partout.
- En C, la casse est **sensible** (`note` ≠ `Note`).

Souhaites-tu que je crée une **fiche d'exercices (10 activités)** où les élèves doivent corriger ou proposer de bons noms de variables selon ces règles ?

C - Affichage avec `printf`

> *La fonction `printf()` est la principale instruction d'affichage du langage C. Elle permet d'écrire du texte, des nombres ou des variables dans la console, souvent pour informer l'utilisateur ou vérifier le comportement d'un programme. Très flexible, elle utilise des spécificateurs de format comme %d, %f, %c ou %s pour afficher différents types de données.*

Objectifs

À la fin de ce chapitre, tu sauras :

- Comprendre à quoi sert `printf()`.
- Afficher du texte et des variables à l'écran.
- Utiliser les principaux **spécificateurs de format** (`%d` , `%f` , `%c` , `%s`).
- Gérer les retours à la ligne et les tabulations.

Principe

`printf()` (abréviation de *print formatted*) est la **fonction principale d'affichage** en langage C. Elle permet d'écrire du texte, des nombres ou des valeurs de variables dans la **console**. Elle fait partie de la bibliothèque standard `stdio.h`, qu'il faut inclure au début du programme :

```
#include <stdio.h>
```

Exemple simple

```
#include <stdio.h>

int main() {
    printf("Bonjour, monde !");
    return 0;
}
```

💡 Le texte à afficher est toujours placé **entre guillemets** `" "`.

Afficher des variables

Pour afficher la valeur d'une variable, on utilise un **spécificateur de format** (`%...`) qui indique à `printf` le **type de donnée** à afficher.

SPÉCIFICATEUR	TYPE AFFICHÉ	EXEMPLE
`%d`	entier (`int`)	`printf("%d", age);`
`%f`	réel (`float`)	`printf("%f", moyenne);`
`%c`	caractère (`char`)	`printf("%c", lettre);`
`%s`	chaîne de caractères (`char[]`)	`printf("%s", nom);`

Exemple :

```
int age = 16;
printf("J'ai %d ans.\n", age);
```

Résultat :

```
J'ai 16 ans.
```

Plusieurs valeurs dans le même texte

Tu peux afficher plusieurs variables dans une même chaîne, en respectant l'ordre des formats :

```
int jour = 13;
char mois[] = "octobre";
int annee = 2025;

printf("Nous sommes le %d %s %d.\n", jour, mois, annee);
```

Résultat :

```
Nous sommes le 13 octobre 2025.
```

Sauts de ligne

Le caractère spécial `\n` sert à **faire un saut de ligne**. Quand tu l'utilises dans un `printf`, cela indique à l'ordinateur de **passer à la ligne suivante** après avoir affiché le texte. Sans `\n`, tout s'afficherait **sur une seule ligne** :

Exemple sans saut de ligne :

```
printf("Bonjour");
printf("Tout le monde");
```

Résultat à l'écran :

```
BonjourTout le monde
```

Exemple avec saut de ligne:

```
printf("Bonjour\n");
printf("Tout le monde\n");
```

Résultat à l'écran :

```
Bonjour
Tout le monde
```

Quelques caractères spéciaux

`printf()` comprend certains **caractères spéciaux** qui modifient la mise en forme du texte :

CODE	SIGNIFICATION	EXEMPLE
`\n`	passage à la ligne	`printf("Bonjour!\n");`
`\t`	tabulation	`printf("Nom\tAge\n");`
`\\`	affiche un antislash `\`	`printf("\\");`
`\"`	affiche une guillemet double	`printf("\"C\"");`

Formater les nombres

- Pour afficher un **nombre réel avec 2 décimales** :

  ```
  float moyenne = 14.357;
  printf("Moyenne : %.2f\n", moyenne);
  ```

  Résultat :

  ```
  Moyenne : 14.36
  ```

Dans l'instruction `printf("Moyenne : %.2f\n", moyenne);`, la chaîne entre guillemets contient du texte et un **spécificateur de format** `%.2f` :

- `%f` indique que l'on veut afficher un **nombre réel** (de type `float` ou `double`) ;
- `.2` précise qu'on veut **afficher deux chiffres après la virgule** ;
- `\n` ajoute un **saut de ligne** à la fin du texte.

Ainsi, si la variable `moyenne` vaut `14.357`, le programme affichera :

```
Moyenne : 14.36
```

Le nombre est automatiquement **arrondi** à deux décimales pour un affichage plus lisible (selon les règles classiques des arrondis).

- Tu peux aussi aligner ou espacer les valeurs :

```
printf("%6d\n", 42);   // Affiche "    42" (avec 4 espaces devant)
printf("%06d\n", 42);  // Affiche "000042" (complète avec des 0)
```

- `%6` indique que le nombre doit aligné à droite dans une zone de 6 caractères de long rempli par des espaces
- `%06` indique que le nombre doit aligné à droite dans une zone de 6 caractères de long rempli par des zéros
- `%d` affiche un entier

Erreurs fréquentes

❌ Oublier les guillemets autour du texte :

```
printf(Salut);   // Erreur
```

✅ Correct :

```
printf("Salut");
```

❌ Ne pas mettre le bon format :

```
int x = 10;
printf("%f", x);   // Mauvais type (int ≠ float)
```

✅ Correct :

```
printf("%d", x);
```

À retenir

- `printf()` affiche du texte ou des valeurs à l'écran.
- Le texte est entre guillemets `" "`.
- Les **spécificateurs de format** permettent d'afficher différents types de données.
- `\n` permet de **passer à la ligne suivante**.
- Il faut **faire correspondre les formats et les types** de variables.

Exemples d'erreurs courantes

Voici quelques **exemples classiques de mauvaise utilisation des modificateurs de format** dans `printf()` et leur **impact concret** sur l'affichage 👇

Mauvais format pour un entier

```c
int age = 16;
printf("%f", age);   // Mauvais format
```

💬 **Problème :** `%f` sert à afficher un nombre réel (`float`), pas un entier. 🎯 **Résultat possible :** affichage d'un **nombre aléatoire ou incohérent**, car `printf` interprète les bits de l'entier comme un réel. Ou `0`. Ca dépend du compilateur. ✅ **Correct :**

```c
printf("%d", age);
```

Mauvais format pour un réel

```c
float moyenne = 13.5;
printf("%d", moyenne);   // Mauvais format
```

💬 **Problème :** `%d` est prévu pour un entier. 🎯 **Résultat :** la valeur affichée n'aura aucun sens (souvent un grand entier négatif). ✅ **Correct :**

```c
printf("%f", moyenne);
```

Mauvais format pour une chaîne de caractères

```c
char nom[] = "Alice";
printf("%c", nom);   // Mauvais format
```

💬 **Problème :** `%c` affiche un seul caractère, alors que `nom` est un tableau (`char[]`). 🎯 **Résultat :** l'affichage est imprévisible (souvent un symbole bizarre). ✅ **Correct :**

```
printf("%s", nom);
```

▸ Trop ou pas assez de variables

```
int a = 5, b = 10;
printf("a = %d, b = %d, c = %d\n", a, b);   // Une variable manquante
```

💬 **Problème :** le dernier `%d` n'a pas de valeur associée. 🎯 **Résultat :** affichage d'un nombre aléatoire, ou plantage du programme. ✅ **Correct :**

```
printf("a = %d, b = %d\n", a, b);
```

▸ Ordre incorrect des formats

```
int x = 5;
float y = 3.14;
printf("x = %f, y = %d\n", x, y);   // Mauvais ordre
```

💬 **Problème :** les variables ne correspondent pas au bon type de format. 🎯 **Résultat :** les valeurs affichées seront totalement fausses. ✅ **Correct :**

```
printf("x = %d, y = %f\n", x, y);
```

▸ Oublier le retour à la ligne

```
printf("Bonjour");
printf("tout le monde");
```

💬 **Problème :** le texte s'enchaîne sans séparation. 🎯 **Résultat :**

```
Bonjourtout le monde
```

✅ **Correct :**

```
printf("Bonjour\n");
printf("tout le monde\n");
```

À retenir

- Chaque **modificateur de format** (`%d` , `%f` , `%c` , `%s` , etc.) correspond à un **type précis**.
- Si le format ne correspond pas au type, le programme affiche des **valeurs incohérentes** (ou plante).
- Il faut toujours vérifier que **le nombre et le type** des formats correspondent exactement à ceux des variables.

💪 C - `printf` exercices

> *Voici quelques exercices sur `printf`*

Parfait 👍 Voici **20 exercices progressifs** pour apprendre et maîtriser `printf()` en deux parties :

- **Série 1 :** afficher du texte et des valeurs simples.
- **Série 2 :** afficher plusieurs variables, formater les nombres, et utiliser les caractères spéciaux (`\n` , `\t` , etc.).

Chaque exercice est clair, autonome, et adapté à des débutants en C.

Exercices – Série 1 : bases de `printf()`

> *A envoyer dans `c/printf/Série 1` .*

▸ Objectifs

Apprendre à afficher du texte, des nombres et des variables avec `printf()` .

▸ Exercice 1 – Ton premier message

Affiche à l'écran :

```
Bonjour, je débute en langage C !
```

▸ Exercice 2 – Phrase avec retour à la ligne

Affiche :

```
Bonjour
Tout le monde
```

💡 Utilise `\n` pour le saut de ligne.

▸ Exercice 3 – Afficher un nombre entier

Déclare une variable `int age = 16;` Affiche :

```
J'ai 16 ans.
```

▸ Exercice 4 – Afficher un réel

Déclare `float moyenne = 14.5;` Affiche :

```
Ma moyenne est de 14.50
```

▸ Exercice 5 – Afficher un caractère

Déclare `char lettre = 'A';` Affiche :

```
La première lettre est A
```

▸ Exercice 6 – Combiner texte et variables

Déclare `char prenom[] = "Alice"; int age = 17;` Affiche :

```
Je m'appelle Alice et j'ai 17 ans.
```

▸ Exercice 7 – Afficher plusieurs entiers

Déclare `int a = 5, b = 10;` Affiche :

```
a = 5 et b = 10
```

▸ Exercice 8 – Tabulations

Affiche un tableau simple à l'aide de `\t` :

```
Nom     Age
Alice   17
Bob     18
```

#Exercices – Série 2 : formats et mise en forme

> *A envoyer dans `c/printf/Série 2` .*

▸ Objectifs

Apprendre à utiliser les formats (`%d` , `%f` , `%c` , `%s`), contrôler l'affichage, et formater les nombres.

▸ Exercice 1 – Affichage avec variables

Déclare `int jour = 13; char mois[] = "octobre"; int annee = 2025;` Affiche :

```
Nous sommes le 13 octobre 2025.
```

▸ Exercice 2 – Nombre avec décimales

Déclare `float prix = 12.5;` Affiche avec **2 chiffres après la virgule** :

```
Prix = 12.50 €
```

▸ Exercice 3 – Arrondi automatique

Déclare `float moyenne = 14.357;` Affiche :

```
Moyenne : 14.36
```

▸ Exercice 4 – Afficher plusieurs lignes avec une seule instruction

Affiche avec un seul `printf` :

```
Ligne 1
Ligne 2
Ligne 3
```

▸ Exercice 5 – Combiner texte et calcul

Déclare `int a = 5, b = 3;` Affiche :

```
5 + 3 = 8
```

💡 Astuce : tu peux calculer directement dans `printf()` → `%d` , `(a + b)` .

▸ Exercice 6 – Espaces réservés

Affiche les nombres suivants alignés à droite sur 5 caractères :

```
  12
 123
1234
```

💡 Utilise `%5d`.

▸ Exercice 7 – Zéros à gauche

Affiche les nombres suivants avec des zéros devant pour qu'ils fassent 4 chiffres :

```
0012
0123
1234
```

💡 Utilise `%04d`.

▸ Exercice 8 – Plusieurs formats

Déclare :

```c
int entier = 42;
float reel = 3.14;
char lettre = 'Z';
char mot[] = "Bonjour";
```

Affiche toutes les variables sur une seule ligne.

▸ Exercice 9 – Utiliser les guillemets et les antislashs

Affiche exactement :

```
Il a dit : "C'est du C !"
Chemin : C:\Users\Alice\
```

▸ Exercice 10 – Ligne de résumé formatée

Déclare :

```c
char prenom[] = "Léo";
int age = 16;
float moyenne = 14.75;
```

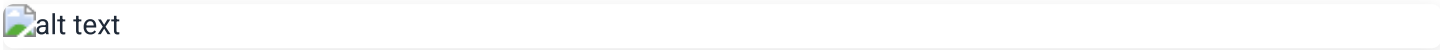
Affiche :

```
Élève : Léo | Âge : 16 | Moyenne : 14.75
```

Calculs des formes géométriques

Crée les programmes C pour calculer l'aire et le périmètre des figures suivantes, en utilisant des noms de variables corrects.

> *A envoyer dans* `c/printf/Formules`.



C - Lecture au clavier avec `scanf`

> *La fonction `scanf` permet de lire des données au clavier, mais elle doit être utilisée avec soin : elle comporte plusieurs pièges (espaces, ``restants, dépassements…). Ce chapitre t'explique comment l'utiliser correctement *et* comment vider le buffer lorsque c'est nécessaire.*

Objectifs

À la fin de ce cours, tu sauras :

- utiliser `scanf` pour lire les principaux types ;
- comprendre le rôle de `&` ;
- choisir les bons formats (`%d` , `%f` , `%s` , …) ;
- éviter les pièges liés au buffer ;
- **vider le buffer proprement** ;
- basculer vers `fgets` quand c'est plus fiable.

Lire les types simples

Lire un entier

```c
int age;
scanf("%d", &age);
```

Lire un flottant

```c
float moyenne;
scanf("%f", &moyenne);
```

Lire un caractère

```c
char lettre;
scanf(" %c", &lettre);
```

💡 L'espace avant `%c` permet d'ignorer le `\n` restant dans le buffer.

Lire une chaîne de caractères (`char[]`)

Sans espaces

```
char nom[30];
scanf("%s", nom);
```

⚠️ `%s` s'arrête au premier espace.

Avec espaces → utiliser `fgets`

```
fgets(nom, sizeof(nom), stdin);
```

Les pièges de `scanf`

Oublier le `&`

❌ `scanf("%d", age);` ✔ `scanf("%d", &age);`

Lire trop de caractères avec `%s`

❌ `scanf("%s", nom);` ✔ `scanf("%29s", nom);`

`%c` lit les retours à la ligne

✔ `scanf(" %c", &lettre);`

Mélanger formats et types → résultats incohérents

Ne jamais faire :

```
scanf("%f", &x);
scanf("%d", &monFloat);
```

`scanf` ne vide jamais le buffer

Si tu tapes :

```
25↵
```

→ `25` est lu → `↵` reste en mémoire → la prochaine lecture peut capter ce `\n`

C'est une source majeure de bugs.

Comment vider le buffer correctement

Il existe plusieurs techniques selon la situation. Voici les 3 **méthodes fiables** à enseigner.

✔ Méthode 1 : Consommer tout ce qui reste jusqu'au prochain `\n`

```c
int c;
while ((c = getchar()) != '\n' && c != EOF) {
    // on vide
}
```

Cette méthode :

- lit caractère par caractère,
- s'arrête quand on atteint le `\n`,
- vide proprement le buffer.

👉 C'est la méthode **classique et recommandée** en C.

✔ Méthode 2 : Appeler `fflush(stdin)` (⚠ non standard)

```c
fflush(stdin);
```

⚠ Important :

- Cela **fonctionne sous Windows / MinGW**,
- mais **n'est pas standard** → peut ne pas fonctionner ailleurs.

👉 À éviter pour un cours strictement ISO C. Mais utile à connaître si tu compiles sous Windows.

✔ Méthode 3 : Lire tout ce qui arrive dans une chaîne temporaire

Utile après une lecture incorrecte :

```c
char tampon[100];
fgets(tampon, sizeof(tampon), stdin);
```

Cela vide proprement **toute la ligne restante**.

▸ Quand vider le buffer ?

1. **Après un** `scanf("%d")` **ou** `scanf("%f")` si tu comptes lire un `char` juste après.
2. **Après une mauvaise saisie** (ex : l'utilisateur tape du texte au lieu d'un nombre).
3. **Avant une lecture avec** `fgets`, si un `scanf` a été utilisé juste avant.
4. **Chaque fois que tu observes un comportement bizarre** où une fonction semble "sauter" une entrée.

Formats les plus utiles

TYPE	FORMAT
`int`	`%d`
`float`	`%f`
`double`	`%lf`
`char`	`" %c"`
chaîne simple	`%s`
chaîne limitée	`%20s`

Utiliser `scanf` avec plusieurs valeurs

```c
int a, b;
scanf("%d %d", &a, &b);
```

Vérifier si la lecture a réussi

```c
if (scanf("%d", &x) != 1) {
    printf("Erreur de saisie\n");
}
```

Alternatives plus sûres

- `fgets()`
- `sscanf()`
- parsing manuel avec `strtol()` ou `strtof()`

À retenir

- `scanf` lit dans tes variables → `&` obligatoire sauf pour les tableaux.
- `%s` est dangereux : le limiter (`"%29s"`) ou utiliser `fgets` .
- `%c` doit presque toujours avoir un espace devant.
- **Scanner laisse toujours des caractères dans le buffer** → il faut parfois le vider.
- Méthode la plus sûre : `while (getchar() ≠ '\n');`

C - Les commentaires

> *En programmation, les commentaires servent à expliquer ton code sans influencer son exécution. Ils permettent de rendre un programme plus lisible, compréhensible et facile à maintenir, aussi bien pour toi que pour les autres développeurs qui le liront plus tard. Bien commenter, c'est une bonne habitude professionnelle.*

Objectifs

À la fin de ce chapitre, tu sauras :

- Comprendre le rôle des commentaires en C.
- Utiliser correctement les deux types de commentaires (`//` et `/* ... */`).
- Savoir **quand et comment** ajouter un commentaire utile.

Qu'est-ce qu'un commentaire ?

Un **commentaire** est une partie du code **ignorée par le compilateur**. Il ne modifie donc **en rien** le comportement du programme : c'est uniquement une **aide pour l'humain**.

Les deux formes de commentaires

1. Commentaire sur une seule ligne

On utilise `//` pour écrire un commentaire court sur une ligne. Tout ce qui suit `//` sur la même ligne est ignoré par le compilateur.

Exemple :

```c
int age = 16; // variable qui contient l'âge de l'utilisateur
```

Tu peux aussi placer un commentaire seul sur une ligne :

```c
// Calcul de la moyenne
float moyenne = (note1 + note2) / 2;
```

▸ 2. Commentaire sur plusieurs lignes

Pour des explications plus longues, on utilise `/* ... */`. Tout ce qui se trouve entre ces symboles est ignoré.

Exemple :

```
/*
Ce programme calcule la moyenne de deux nombres.
Il affiche ensuite le résultat à l'écran.
*/
float moyenne = (note1 + note2) / 2;
```

💡 Utile pour commenter des **blocs de code**, des **sections** ou même pour **désactiver temporairement** une partie du programme.

Exemple complet

```
#include <stdio.h>

int main() {
    // Entrées
    int note1 = 12;
    int note2 = 16;

    /* Traitements
       On utilise un type float pour garder les décimales */
    float moyenne = (note1 + note2) / 2.0;

    // Sorties
    printf("Moyenne : %.2f\n", moyenne);

    return 0; // Fin du programme
}
```

Bonnes pratiques

✅ **À faire :**

- Expliquer la logique ou l'objectif d'une section de code.
- Commenter les lignes importantes ou complexes.
- Mettre à jour les commentaires quand tu modifies le code.
- Garder les commentaires courts, clairs et utiles.

❌ **À éviter :**

- Répéter ce que le code dit déjà :

```
int x = 5; // on met 5 dans x  ❌ inutile
```

- Trop commenter : un code trop annoté devient illisible.
- Laisser des vieux commentaires qui ne correspondent plus au code.

Quand commenter ?

- Avant un **bloc de code** pour expliquer son but.
- À côté d'une ligne **importante** pour préciser un calcul ou une astuce.
- En haut d'un fichier pour décrire **l'objectif global du programme** (titre, auteur, date, description).

Exemple :

```
/*
Programme : calcul de la surface d'un rectangle
Auteur : Martin
Date : 05/11/2025
Description : lit la longueur et la largeur, calcule la surface et l'affiche.
*/
```

Ignorer du code

Tu peux aussi mettre un code en commentaire pour le désactiver, pour qu'il soit ignoré par le compilateur, sans devoir l'effacer de ton programme.

À retenir

- Les commentaires servent à **expliquer le code**, pas à le remplacer.
- `//` → pour une seule ligne ; `/* ... */` → pour plusieurs lignes.
- Ils n'ont **aucun effet sur le programme** : ils sont ignorés à la compilation.
- De bons commentaires rendent ton code **plus clair, plus propre et plus professionnel**.
- Les commentaires peuvent aussi à **désactiver du code** sans le supprimer de ton programme

C - Les structures (`struct`)

> *Une structure (ou *struct*) permet de **regrouper plusieurs variables** sous un même nom. Une `struct` représenter des données plus complexes qu'un simple entier ou qu'un `float`. Avec une structure, tu peux modéliser un élève, un produit, un point, une date… en associant plusieurs informations dans un même ensemble logique.*

Ce chapitre t'apprendra à créer, utiliser et comprendre les *struct* en C.

Objectifs

À la fin de cette leçon, tu sauras :

- Définir une structure avec le mot-clé `struct`.
- Déclarer et utiliser des variables de type structure.
- Accéder aux champs d'une structure avec l'opérateur `.`
- Utiliser `typedef` pour simplifier l'écriture.
- Comprendre comment les structures sont stockées en mémoire.

Qu'est-ce qu'une structure ?

Une structure regroupe plusieurs variables (appelées **champs**) qui peuvent être de types différents.

Exemple : représenter un élève avec

- son nom,
- son âge,
- sa moyenne.

Sans `struct`, tu aurais trois variables séparées. Avec `struct`, tu obtiens un seul "bloc" cohérent.

Définir une structure

Exemple simple

```
typedef struct {
    char nom[30];
    int age;
    float moyenne;
} Etudiant;
```

Ici :

- `nom` est un tableau de 30 caractères,
- `age` est un entier,
- `moyenne` est un `float`.

Tu viens de créer **un nouveau type personnalisé** : `struct Etudiant`.

`typedef` permet d'utiliser ta structure plus simplement par la suite (cela te permet de faire `Etudiant manu = ...` au lieu de `struct Etudiant manu= ...` quand tu déclares tes variables).

Simpl

Déclarer et utiliser une structure

```
Etudiant e1;

strcpy(e1.nom, "Alice");
e1.age = 17;
e1.moyenne = 14.5;

printf("%s a %d ans et %.2f de moyenne.\n", e1.nom, e1.age, e1.moyenne);
```

Accès aux champs

On utilise l'opérateur `.` :

```
e1.nom
e1.age
e1.moyenne
```

Créer et initialiser en une ligne

```
struct Etudiant e2 = {"Lucas", 18, 12.8};
```

Structures contenant plusieurs tableaux

Oui, c'est possible. Exemple :

```
typedef struct {
    char nom[30];
    char prenom[40];
    int age;
} Personne;
```

Chaque tableau peut contenir une chaîne de caractères distincte.

Lire des données dans une structure

```
Personne p;

printf("Nom : ");
scanf("%s", p.nom);

printf("Prénom : ");
scanf("%s", p.prenom);

printf("Âge : ");
scanf("%d", &p.age);
```

💡 Les tableaux (`char[]`) sont automatiquement passés par référence → pas de `&` .

Structures et mémoire

Une structure regroupe ses champs **de manière contiguë en mémoire**.

Exemple :

```
struct Etudiant
{
    char nom[30];   ← 30 octets
    int age;        ← 4 octets
    float moyenne;  ← 4 octets
}
```

La taille totale est au moins 38 octets (souvent un peu plus à cause de l'alignement mémoire).

Structures et fonctions

Une structure peut être passée à une fonction **par valeur** ou **par référence**.

▸ Par valeur (copie)

```
void afficher(Etudiant e) {
    printf("%s (%d ans)\n", e.nom, e.age);
}
```

▸ Par référence (pointeur)

```
void modifier(Etudiant *e) {
    e->age += 1;
}
```

💡 On utilise `→` pour accéder aux champs via un pointeur.

À retenir

- Une structure regroupe plusieurs données liées entre elles.
- On accède aux champs avec `.` ou `→` selon les cas.
- `typedef` permet de créer un type plus simple à utiliser.
- Les tableaux et chaînes dans une structure sont tout à fait possibles.
- Une structure peut être passée par valeur (copie) ou par référence (adresse).

Erreurs courantes à corriger

- Oublier le `;` à la fin de la définition de la structure.
- Confondre tableau de `char` et chaîne "classique".
- Mélanger `.` et `→`.

- Tenter de faire `p.nom = "Alice";` au lieu de `strcpy`.

C - Les conditions

> *Dans la programmation en C, les conditions permettent de contrôler le comportement de votre programme en fonction des événements ou des valeurs mesurées. Elles permettent de prendre des décisions et de déclencher des actions différentes selon les situations. Dans cet article, nous allons explorer les instructions conditionnelles `if`, `else if`, et `else`, ainsi que les opérateurs logiques `&&` (ET) et `||` (OU), pour vous aider à mieux gérer les flux d'exécution dans vos projets.*

Les Bases des Conditions : if, else if, et else

Utilisation de `if`

L'instruction `if` permet de tester une condition et d'exécuter un bloc de code seulement si cette condition est vraie. Si la condition est fausse, le programme ignore le bloc de code.

Syntaxe de base :

```
if (condition) {
    // Code à exécuter si la condition est vraie
}
```

Exemple : Réussir son interro.

```

```

Utilisation de `else`

L'instruction `else` est un cas de repli qui s'exécute si aucune des conditions précédentes (`if` ou `else if`) n'est vraie. Elle est souvent utilisée pour définir une action par défaut.

Syntaxe de base :

```
if (condition1) {
    // Code si condition1 est vraie
} else if (condition2) {
    // Code si condition2 est vraie
} else {
    // Code si aucune condition n'est vraie
}
```

Exemple:

```c
#include <stdio.h>

void main() {
    int note = 13;

    if (note >= 10) {
        printf("Bravo, tu as réussi !\n");
    }
    else {
        printf("Dommage, tu as échoué.\n");
    }

}
```

Utilisation de `else if`

L'instruction `else if` permet de tester une condition supplémentaire si la première condition (`if`) est fausse. Vous pouvez chaîner plusieurs `else if` pour tester différentes conditions.

Syntaxe de base :

```c
#include <stdio.h>

void main() {
    int nombre = 1;

    if (nombre > 0) {
        printf("Le nombre est positif.\n");
    }
    else if (nombre < 0) {
        printf("Le nombre est négatif.\n");
    }
    else {
        printf("Le nombre est nul.\n");
    }

}
```

Opérateurs Logiques : `&&` (ET) et `||` (OU)

Pour combiner plusieurs conditions dans une même instruction `if` , vous pouvez utiliser les opérateurs logiques `&&` (ET) et `||` (OU).

Utilisation de `&&` (ET)

L'opérateur `&&` est utilisé lorsque vous voulez que **plusieurs conditions soient vraies en même temps** pour exécuter le code.

Exemple : vérifier si un nombre est compris entre 0 et 100

```c
#include <stdio.h>

int main() {
    int nombre;

    printf("Entre un nombre : ");
    scanf("%d", &nombre);

    if (nombre >= 0 && nombre <= 100) {
        printf("Le nombre est entre 0 et 100.\n");
    }
    else {
        printf("Le nombre n'est pas dans l'intervalle.\n");
    }

    return 0;
}
```

Utilisation de `||` (OU)

L'opérateur `||` est utilisé lorsque vous voulez que **au moins une des conditions soit vraie** pour exécuter le code.

```c
#include <stdio.h>

int main() {
    int nombre;

    printf("Entre un nombre : ");
    scanf("%d", &nombre);

    if (nombre < 0 || nombre > 100) {
        printf("Le nombre est en dehors de l'intervalle 0-100.\n");
    }
    else {
        printf("Le nombre est entre 0 et 100.\n");
    }

    return 0;
}
```

Explication

- L'expression `(nombre < 0 || nombre > 100)` est **vraie** si le nombre est plus petit que 0 **ou** plus grand que 100.
- Si le nombre est 50 → faux (dans l'intervalle).
- Si le nombre est -5 → vrai.
- Si le nombre est 150 → vrai.

Utiliser des parenthèses

💡 **Astuce :** Utilise toujours des **parenthèses** pour rendre les conditions plus lisibles :

```
if ((a > 0) || (b < 10))
```

Conclusion

Les conditions `if`, `else if`, `else`, ainsi que les opérateurs logiques `&&` et `||`, sont essentiels pour prendre des décisions dans vos programmes. En combinant ces structures de contrôle, vous pouvez créer des comportements complexes et adaptés aux situations de vos projets, qu'il s'agisse de vérifier plusieurs capteurs ou de déclencher des actions en fonction d'événements spécifiques. Prendre le temps de bien comprendre ces concepts est crucial pour développer des programmes robustes et fonctionnels.

💪 C - Les conditions (exercices)

> *Des exercices sur les conditions*

Voici une série de **10 exercices progressifs** pour travailler les **conditions (`if` , `else` , `else if`)** en langage C.

Objectifs

Ces exercices te permettront de :

- Comprendre le fonctionnement des structures conditionnelles (`if` , `else if` , `else`).
- Utiliser les opérateurs de comparaison (`==` , `≠` , `<` , `>` , `≤` , `≥`).
- Combiner plusieurs conditions avec les opérateurs logiques (`&&` , `||`).
- Développer ta logique de raisonnement conditionnel.

Exercice 1 – Positif ou négatif

Demande à l'utilisateur un nombre entier. Affiche :

```
Le nombre est positif.
```

ou

```
Le nombre est négatif.
```

Exercice 2 – Pair ou impair

Lis un nombre et affiche s'il est **pair** ou **impair** à l'aide de l'opérateur `%` . 💡 Rappel : un nombre est pair si `nombre % 2 == 0` .

Exercice 3 – Majeur ou mineur

Demande l'âge de l'utilisateur. Affiche :

```
Tu es majeur.
```

ou

```
Tu es mineur.
```

Exercice 4 – Meilleure note

Demande deux notes (sur 20) à l'utilisateur. Affiche la plus grande des deux :

```
La meilleure note est 14.
```

Exercice 5 – Note et appréciation

Demande une note sur 20 et affiche une appréciation selon le résultat :

- 0 à 9 → "Insuffisant"
- 10 à 13 → "Satisfaisant"
- 14 à 17 → "Bien"
- 18 à 20 → "Excellent"

💡 Utilise plusieurs `else if`.

Exercice 6 – Température correcte

Demande la température (en °C). Si elle est entre 18 et 25 inclus, affiche "Température normale". Sinon, affiche "Température anormale".

Exercice 7 – Accès autorisé

Demande un **code secret** (nombre entier). Si le code vaut `1234`, affiche "Accès autorisé", sinon "Accès refusé".

Exercice 8 – Admis ou non (avec deux conditions)

Demande la **note** et le **taux de présence** (en %). L'élève est admis s'il a **au moins 10 et au moins 80 % de présence**. Sinon, affiche "Refusé".

💡 Utilise l'opérateur logique `&&`.

Exercice 9 – Période de l’année

Demande un mois (nombre entre 1 et 12). Affiche la saison correspondante :

- 12, 1, 2 → Hiver
- 3, 4, 5 → Printemps
- 6, 7, 8 → Été
- 9, 10, 11 → Automne

Exercice 10 – Calculatrice simple

Demande à l’utilisateur deux nombres et une opération (`+` , `-` , `*` , `/`). Affiche le résultat selon l’opération choisie. 💡 Utilise une structure `if ... else if ... else` avec des comparaisons sur un `char` .

Bonus – Pour s’entraîner davantage

- Modifie les programmes pour gérer **les cas d’erreur** (division par zéro, mauvaise saisie, etc.).
- Combine les opérateurs `&&` et `||` pour créer des conditions plus complexes.
- Ajoute des messages plus détaillés pour améliorer la compréhension de l’utilisateur.

C - Les fonctions

> *Une **fonction** est un **bloc de code réutilisable** qui exécute une tâche précise. Plutôt que de répéter les mêmes instructions plusieurs fois, on les regroupe dans une fonction, qu'on peut ensuite **appeler** à différents endroits du programme. Les fonctions permettent d'organiser le code, de le rendre plus **lisible**, plus **court** et plus **facile à corriger**.*

Objectifs

À la fin de ce chapitre, tu sauras :

- Définir et appeler une fonction en C.
- Comprendre la notion de **paramètres** et de **valeur de retour**.
- Utiliser la bonne **structure d'une fonction**.
- Faire la différence entre la **fonction principale** (`main`) et les fonctions que tu crées.

Structure d'une fonction

Une fonction en C suit une structure très précise :

```
type_de_retour nom_de_fonction(paramètres) {
    // instructions
    return valeur;   // facultatif selon le type
}
```

- **type_de_retour** : indique le type de la valeur renvoyée (`int` , `float` , `char` , `void` ...).
- **nom_de_fonction** : choisi par le programmeur, selon sa fonction.
- **paramètres** : variables reçues par la fonction.
- **return** : renvoie une valeur au programme principal (sauf si la fonction est de type `void`).

Une fonction simple sans paramètres

```c
#include <stdio.h>

void direBonjour() {
    printf("Bonjour à tous !\n");
}

int main() {
    direBonjour();  // appel de la fonction
    return 0;
}
```

💡 `void` signifie que la fonction **ne renvoie rien**.

Ici, `direBonjour()` est **appelée depuis** `main()` pour exécuter son code.

Fonction avec paramètres

```c
#include <stdio.h>

void afficherSomme(int a, int b) {
    printf("Somme = %d\n", a + b);
}

int main() {
    afficherSomme(5, 3);
    return 0;
}
```

Ici, `a` et `b` sont des **paramètres** : la fonction les reçoit, calcule leur somme et l'affiche.

Fonction qui renvoie une valeur

```c
#include <stdio.h>

int carre(int nombre) {
    return nombre * nombre;
}

int main() {
    int resultat = carre(6);
    printf("Le carré de 6 vaut %d\n", resultat);
    return 0;
}
```

💡 Le mot-clé `return` renvoie la valeur calculée (`36` ici) au programme principal.

Déclaration avant `main`

En C, une fonction doit être **connue du compilateur** avant d'être utilisée. Si tu écris la fonction **après** `main()`, tu dois **la déclarer** avant :

```c
#include <stdio.h>

int carre(int nombre);   // ← déclaration (prototype)

int main() {
    printf("%d\n", carre(5));
    return 0;
}

int carre(int nombre) {  // ← définition
    return nombre * nombre;
}
```

Le **prototype** indique simplement au compilateur :

> *"Il existe une fonction nommée `carre` qui prend un `int` et renvoie un `int`."*

Je trouve qu'il est plus lisible de placer la fonction `main` en début de programme et les autres fonctions après la fonction `main` : on sait directement où commence notre programme. Dans ce cas, il faut penser à déclarer tes fonctions avant... mais c'est un tout petit prix à payer...

Plusieurs fonctions

Tu peux évidemment écrire plusieurs fonctions dans le même programme :

```c
#include <stdio.h>

void afficherMessage() {
    printf("Programme de calcul\n");
}

int somme(int a, int b) {
    return a + b;
}

int main() {
    afficherMessage();
    printf("Résultat : %d\n", somme(4, 7));
    return 0;
}
```

À retenir

- Une **fonction** = un **nom + un rôle précis + un bloc d'instructions**.
- Les **paramètres** permettent de transmettre des valeurs.
- `return` renvoie un résultat à celui qui a appelé la fonction.
- `void` → pas de valeur renvoyée.
- Toujours **déclarer** une fonction avant de l'utiliser si elle se trouve après `main()`.
- Les fonctions rendent ton code **plus clair, plus structuré et réutilisable**.

C - Passage de paramètres par valeur et par référence

> *Quand on appelle une fonction en C, on peut lui **transmettre des valeurs** à traiter. Mais il y a uen différence importante à prendre en compte selon que tu veuilles modifier ces valeurs ou pas.*

Introduction

Selon la manière dont on transmet les valeur à une fonction, celle-ci peut :

- soit **travailler sur une copie** de la variable (sans modifier l'originale),
- soit **modifier directement la variable d'origine**.

C'est la différence entre le **passage par valeur** et le **passage par référence**.

Objectifs

À la fin de ce chapitre, tu sauras :

- Faire la différence entre **passage par valeur** et **passage par référence**.
- Comprendre pourquoi on utilise `&` pour récupérer l'adresse d'une variable.
- Identifier les **types automatiquement passés par référence**.
- Savoir quand utiliser l'une ou l'autre méthode.

Passage par valeur

Principe

Lorsqu'une variable est passée **par valeur**, la fonction reçoit **une copie** de cette variable. Les modifications faites dans la fonction **ne changent pas** la variable originale.

Exemple

```c
#include <stdio.h>

void incrementer(int x) {
    x = x + 1;  // modifie seulement la copie
}

int main() {
    int nombre = 10;
    incrementer(nombre);
    printf("Valeur après appel : %d\n", nombre);
    return 0;
}
```

Résultat

```
Valeur après appel : 10
```

La fonction a modifié **la copie**, jamais la variable d'origine.

Passage par référence

Principe

Dans ce cas, la fonction reçoit **l'adresse mémoire** de la variable, grâce au symbole `&`. Elle peut donc **modifier directement la valeur originale**.

Exemple

```c
#include <stdio.h>

void incrementer(int *x) {
    *x = *x + 1;  // modifie la véritable variable
}

int main() {
    int nombre = 10;
    incrementer(&nombre);
    printf("Valeur après appel : %d\n", nombre);
    return 0;
}
```

▸ Résultat

```
Valeur après appel : 11
```

Types passés automatiquement par référence

En C, certains types sont **toujours passés par référence**, même si tu ne mets pas explicitement de pointeur. Ce n'est pas une règle magique du langage, mais une conséquence directe de la façon dont ces types sont **stockés en mémoire**.

▸ Les tableaux (`int tab[]` , `char nom[]` , etc.)

Quand tu passes un tableau, la fonction reçoit **l'adresse du premier élément**. Le tableau **n'est jamais copié**.

```c
void afficherPremier(int t[]) {
    printf("%d\n", t[0]);
}

int main() {
    int valeurs[3] = {4, 7, 2};
    afficherPremier(valeurs); // pas besoin de &
    return 0;
}
```

➡️ Toute modification dans la fonction modifie le tableau d'origine.

▸ Les chaînes de caractères (`char nom[]`)

Une chaîne est un **tableau de char**, donc même logique :

```c
void mettreEnMaj(char s[]) {
    s[0] = 'A';   // modifie l'original
}
```

➡️ Aucun `&` nécessaire, la chaîne est déjà une adresse.

▸ Les pointeurs

Si tu passes un pointeur, tu passes déjà une **adresse mémoire**.

```c
void changer(int *p) {
    *p = 99;
}
```

➡️ Directement par référence.

Comparaison des deux approches

MODE DE PASSAGE	EXEMPLE D'APPEL	MODIFIE LA VARIABLE ORIGINALE ?
Par valeur	`incrementer(x)`	❌ non
Par référence (via pointeur)	`incrementer(&x)`	✔ oui
Tableau	`fonction(tab)`	✔ oui (toujours)
Chaîne de caractères	`fonction(nom)`	✔ oui (toujours)

À retenir

- **Par valeur** → la fonction reçoit une **copie** → la variable originale ne change pas.
- **Par référence** → la fonction reçoit **une adresse** → la variable originale peut changer.
- Les **tableaux** et **chaînes de caractères** sont **toujours** transmis par référence.
- On utilise `*` et `&` pour manipuler les adresses et les valeurs pointées.

C - Les fonctions - Exercices 💪

> *Voici une **feuille complète de 15 exercices progressifs** sur le passage de paramètres **par valeur**, **par référence**, et les types transmis **par adresse** (tableaux, chaînes).*

Objectifs

Ces exercices te permettent de :

- comprendre la différence entre passage par valeur et par référence ;
- utiliser `&` et `*` correctement ;
- modifier des variables dans une fonction ;
- manipuler des tableaux et chaînes passés par adresse.

▸ Effet d'un paramètre passé par valeur

Écris une fonction `incrementer(int x)` qui augmente `x` de 1. Dans `main`, montre que la variable originale **ne change pas**.

▸ Effet d'un paramètre passé par référence

Écris une fonction `incrementerRef` qui augmente une variable de 1. Dans `main`, montre que la variable originale **est modifiée**.

▸ Doubler une valeur (référence)

Écris une fonction `doubler` qui multiplie une variable par 2. Test :

```
x = 7  → doubler →  x = 14
```

▸ Mettre un nombre à zéro (référence)

Écris une fonction `raz` qui met une variable à 0.

▸ Comparer deux entiers (valeur)

Écris une fonction `maximum` qui renvoie le plus grand. Montre qu'elle **ne modifie pas** les variables envoyées.

▸ Comparer deux entiers (référenc

Même exercice que le précédent, mais cette fois ta fonction `trier` doit mettre dans `a` le minimum et dans `b` le maximum.

Exemple : `a = 12, b = 5 → a = 5, b = 12`.

▸ Échanger deux valeurs (swap)

Écris une fonction `echanger`.

Après l'appel, les valeurs doivent être inversées.

Test : `a = 5, b = 8 → a = 8, b = 5`.

▸ Mettre en majuscule le premier caractère d'une chaîne

Écris une fonction `majPremier` qui transforme `texte[0]` en majuscule. Test : `"alice"` → `"Alice"`.

▸ Compter les voyelles dans une chaîne

Écris une fonction `int compterVoyelles`. Appelle-la depuis `main` et affiche le résultat.

▸ Remplacer tous les 'a' par des '@' dans une chaîne

Écris une fonction `remplacer` qui modifie directement la chaîne. Test : `"banane"` → `"b@n@ne"`.

▸ Calculer la somme d'un tableau

Écris une fonction `int somme` qui renvoie la somme des cases. Montre dans `main` que le tableau n'est **pas modifié** (passage par référence mais lecture seulement).

▸ Mettre toutes les cases d'un tableau à zéro

Écris une fonction `vider` qui met toutes les valeurs à 0. Test : `{4, 1, 9}` → `{0, 0, 0}`.

▸ Trouver la plus grande valeur et renvoyer sa position

Écris une fonction `indexMax`. Elle renvoie l'indice de la plus grande valeur.

Test : `{4, 9, 7}` → retourne `1`.

⚠️ Le tableau est passé par référence, mais **tu ne dois rien modifier**.

▸ Échanger deux cases d'un tableau

Écris une fonction :

```
void echangerCases(int tab[], int i, int j);
```

Elle échange les cases `tab[i]` et `tab[j]`. Test : `tab = {1, 2, 3, 4}`, `i = 1`, `j = 3` → `{1, 4, 3, 2}`.

À retenir

- Les **variables simples** sont passées par valeur.
- Les **pointeurs**, **tableaux** et **chaînes** sont passés par référence.
- Le passage par référence permet de **modifier la variable originale**.
- Toutes les fonctions avec des `int *`, `char[]` ou `int tab[]` agissent directement sur les données du `main`.

C - Comment documenter correctement une fonction

> *Documenter une fonction permet de comprendre rapidement **à quoi elle sert**, quels paramètres elle reçoit et ce qu'elle renvoie. Une bonne documentation rend ton code plus lisible, plus professionnel et beaucoup plus facile à maintenir.*

Introduction

Objectifs

À la fin de ce chapitre, tu sauras :

- Écrire une documentation courte et claire pour une fonction.
- Utiliser le format de style *javadoc* (description + tags).
- Décrire les paramètres et la valeur retournée.
- Reconnaître ce qu'il faut commenter ou non.

Format recommandé (style javadoc)

Voici le modèle que tu utiliseras pour documenter tes fonctions :

```
/**
 * Phrase courte : dit ce que fait la fonction, en une seule phrase.
 *
 * Description plus longue (facultatif) : précise le fonctionnement,
 * les cas particuliers, les effets de bord, etc.
 *
 * @param x  description du paramètre x
 * @param y  description du paramètre y
 * @return   description de la valeur retournée
 */
int exemple(int x, int y) {
    return x + y;
}
```

Fonction simple sans paramètres

```c
/**
 * Affiche un message de bienvenue.
 *
 * Utilise printf pour écrire un texte simple dans la console.
 */
void afficherMessage() {
    printf("Bienvenue !\n");
}
```

Fonction avec paramètres

```c
/**
 * Calcule la moyenne de deux nombres réels.
 *
 * Les deux valeurs sont additionnées, puis divisées par deux.
 *
 * @param a  premier nombre
 * @param b  deuxième nombre
 * @return   la moyenne de a et b
 */
float moyenne(float a, float b) {
    return (a + b) / 2;
}
```

Bonnes pratiques

✔ À faire

- Toujours commencer par **une phrase courte**, terminée par un point.
- Rédiger ensuite (facultatif) une description plus longue.
- Décrire brièvement **chaque paramètre** avec `@param`.
- Décrire la valeur retournée avec `@return`.
- Rester cohérent dans tout le projet.
- Documenter les **effets de bord** (variables modifiées, tableau changé…).

❌ À éviter

- Décrire ligne par ligne ce que fait le code.
- Écrire des commentaires trop longs (ce n'est pas un roman).

- Oublier de mettre à jour la documentation après une modification.
- Documenter l'évident :

```
int x;  // variable x
```

Mauvais vs bon commentaire

❌ Mauvais

```
// calcule une somme
int f(int a, int b) {
    return a + b;
}
```

✔ Bon

```
/**
 * Calcule la somme de deux entiers.
 *
 * Retourne la valeur obtenue en additionnant a et b.
 *
 * @param a  premier entier
 * @param b  deuxième entier
 * @return   la somme de a et b
 */
int somme(int a, int b) {
    return a + b;
}
```

À retenir

- Une bonne documentation doit être **courte, utile et précise**.
- Le modèle javadoc aide à structurer ce que tu écris.
- On documente surtout **le but**, les **paramètres**, les **retours**, et les éventuels **effets de bord**.
- La qualité du commentaire reflète la qualité du code.

C - Les boucles

> *Quand tu veux qu'un programme **répète plusieurs fois** une même action, tu utilises une **boucle**. C'est un mécanisme essentiel en programmation : il permet d'automatiser des tâches répétitives sans devoir copier-coller le même code plusieurs fois.*

Quand tu veux qu'un programme **répète plusieurs fois** une même action, tu utilises une **boucle**. C'est un mécanisme essentiel en programmation : il permet d'automatiser des tâches répétitives sans devoir copier-coller le même code plusieurs fois.

Par exemple, si tu veux afficher les nombres de 1 à 10, tu pourrais écrire dix fois `printf()`. Mais une **boucle** permet de le faire en **trois lignes** : le programme recommence automatiquement tant qu'une condition est vraie.

Pourquoi utiliser une boucle ?

Une boucle sert à :

- **répéter une instruction** ou un bloc d'instructions ;
- **éviter les répétitions manuelles** de code ;
- **rendre le programme plus compact et plus clair** ;

Les boucles sont donc **indispensables** pour traiter des données, parcourir des tableaux, ou effectuer des calculs successifs.

Les trois grands types de boucles en C

1. `while` – "tant que" → On répète une action **tant qu'une condition est vraie**. Exemple : lire des nombres tant qu'ils sont positifs.
2. `do ... while` – "faire ... tant que" → Même principe, mais la condition est testée **à la fin**. Cela garantit **au moins une exécution** du bloc.
3. `for` – "pour" → La boucle la plus compacte pour répéter un nombre précis de fois. Exemple : afficher les nombres de 1 à 10.

Exemple simple

```c
int i = 1;
while (i <= 5) {
    printf("Tour %d\n", i);
    i++;
}
```

Résultat :

```
Tour 1
Tour 2
Tour 3
Tour 4
Tour 5
```

Le programme **répète les instructions** entre `{` et `}` tant que `i ≤ 5`. Quand `i` devient 6, la condition est fausse, la boucle s'arrête.

Bonnes pratiques

- Ne crée **jamais une boucle infinie involontairement** : vérifie toujours qu'une condition finit par devenir fausse.
- Utilise des **noms de variables clairs** (comme `compteur`, `index`, `i` pour un compteur).
- Commente ton code si la condition de la boucle n'est pas évidente.
- Choisis le **type de boucle** adapté à la situation :
  - `while` → tu ne sais pas combien de fois il faudra répéter.
  - `for` → tu sais combien de fois tu veux répéter.
  - `do ... while` → tu veux exécuter **au moins une fois**.

À retenir

- Une **boucle** répète un bloc d'instructions tant qu'une condition est vraie.
- Il existe **trois types principaux** : `while`, `do ... while`, et `for`.
- Une boucle a toujours :
  - une **condition** (quand s'arrêter),
  - un **bloc d'actions** (quoi faire),
  - souvent une **variable de contrôle** (compteur ou état).

Dans les prochains articles, tu apprendras à utiliser chaque type de boucle en détail, avec des exemples concrets et des exercices.

C - Les boucles `for`

> *En programmation, la boucle `for` est la plus utilisée lorsqu'on **sait d'avance combien de fois** une action doit être répétée. Elle est compacte, claire, et très pratique pour parcourir des séquences de nombres ou de positions dans un tableau.*

Principe

La boucle `for` permet d'exécuter un bloc d'instructions un certain nombre de fois, en **initialisant**, **testant**, puis **modifiant** une variable de contrôle à chaque tour.

Sa syntaxe est la suivante :

```
for (initialisation; condition; mise à jour) {
    // instructions à répéter
}
```

Ces trois parties se comportent comme suit :

1. **initialisation** → exécutée **une seule fois** avant le début de la boucle ;
2. **condition** → testée avant **chaque tour** (si elle devient fausse, la boucle s'arrête) ;
3. **mise à jour** → exécutée **après chaque tour**.

Exemple de base

```c
for (int i = 1; i <= 5; i++) {
    printf("Tour %d\n", i);
}
```

Résultat :

```
Tour 1
Tour 2
Tour 3
Tour 4
Tour 5
```

Décryptage

- `int i = 1` : on crée un compteur `i` qui démarre à 1 ;
- `i ≤ 5` : la boucle continue **tant que** `i` est inférieur ou égal à 5 ;

- `i++` : à la fin de chaque tour, on **incrémente** `i` de 1.

Quand `i` devient 6, la condition est fausse et la boucle s'arrête.

Exemple concret

```c
int somme = 0;
for (int i = 1; i <= 10; i++) {
    somme += i;  // ajoute i à la somme
}
printf("Somme = %d\n", somme);
```

Ce programme calcule la somme des nombres de 1 à 10.

Variante : décrémentation

Tu peux aussi faire une boucle qui **compte à rebours** :

```c
for (int i = 10; i >= 1; i--) {
    printf("%d\n", i);
}
printf("Décollage !\n");
```

Bonnes pratiques

- Utilise `for` lorsque **le nombre d'itérations est connu** à l'avance.
- Choisis un **nom de compteur clair** (`i` , `j` , `k` pour des boucles imbriquées, ou `compteur` , `index` pour plus de clarté).
- N'oublie **jamais la condition d'arrêt**, sinon tu crées une boucle infinie.
- Évite de modifier la variable de contrôle **dans le bloc** : garde-la uniquement dans la partie `mise à jour` .

À retenir

- `for` est idéale pour répéter une action **un nombre précis de fois**.
- Elle regroupe en une seule ligne :
  - **l'initialisation**,
  - **le test de condition**,
  - **la mise à jour**.

- C’est la boucle la plus pratique pour **parcourir des tableaux ou générer des suites de nombres**.

Dans la suite, tu verras les boucles `while` et `do ... while`, qui servent quand le nombre de répétitions **n’est pas connu d’avance**.

C - Les boucles `for` - Exercices 💪

> *Voici **10 exercices progressifs en C** pour apprendre à maîtriser les **boucles `for`**, adaptés à un cours débutant mais exigeant. Chaque exercice inclut un objectif clair et une consigne précise.*

1) Affichage simple de nombres

Objectif : comprendre la structure de base d'une boucle `for`. **Consigne :** afficher les nombres de **1 à 10** sur une seule ligne, séparés par des espaces.

---

2) Compter à rebours

Objectif : travailler la décrémentation dans un `for`. **Consigne :** afficher les nombres de **10 à 1**.

---

3) Afficher les nombres pairs

Objectif : utiliser une condition dans une boucle `for`. **Consigne :** afficher tous les **nombres pairs entre 1 et 50**.

---

4) Somme des N premiers nombres

Objectif : accumuler une valeur dans une variable. **Consigne :** demander un nombre `n` à l'utilisateur et calculer la **somme des nombres de 1 à n**.

---

5) Compter les multiples

Objectif : analyser une séquence. **Consigne :** afficher tous les **multiples de 7 entre 1 et 200**, et afficher ensuite combien il y en avait.

---

6) Table de multiplication

Objectif : produire une sortie structurée. **Consigne :** demander un entier et afficher sa **table de multiplication (1 → 10)**.

---

7) Calcul de factorielle

Objectif : utiliser une variable multiplicative. **Consigne :** demander un nombre entre 1 et 12 et calculer `n!` (factorielle de n).

---

8) Identifier le minimum et le maximum

Objectif : parcourir plusieurs valeurs entrées par l'utilisateur. **Consigne :** demander **10 nombres** et déterminer le **minimum** et le **maximum**.

---

9) Triangle d'étoiles (motif ASCII)

Objectif : utiliser une boucle `for` imbriquée. **Consigne :** afficher un triangle comme ceci (pour une hauteur `n`) :

```
*
**
***
****
```

---

10) Nombres premiers dans un intervalle

Objectif : implanter une logique plus complexe avec boucles imbriquées. **Consigne :** afficher tous les **nombres premiers entre 1 et 200**.

Utiliser une boucle `for` externe pour parcourir les nombres et une boucle interne pour tester la divisibilité.

C - #include

> *En langage C, un programme ne se limite pas à un simple fichier. Dès que la taille d'un projet augmente, il devient indispensable de **structurer le code**, de **réutiliser des fonctions existantes** et de **séparer les responsabilités***

Le langage C repose pour cela sur un système de **bibliothèques** et de **fichiers organisés** (`.h` et `.c`), reliés entre eux grâce à la directive `#include` . Comprendre ce mécanisme est fondamental : il permet non seulement d'utiliser les fonctions standard du langage, mais aussi de créer ses propres bibliothèques, claires, maintenables et professionnelles.

Ce chapitre explique comment fonctionne `#include` , à quoi servent les librairies, et comment organiser correctement un programme C en plusieurs fichiers.

À quoi servent les bibliothèques en C ?

En C, **on ne réécrit pas tout soi-même**. Le langage met à disposition des **bibliothèques** : des ensembles de fonctions déjà écrites, testées et optimisées.

Exemples de fonctions fournies par des bibliothèques :

- `printf()` → affichage à l'écran
- `scanf()` → lecture clavier
- `sqrt()` → racine carrée
- `strlen()` → longueur d'une chaîne

Ces fonctions **n'existent pas par magie** : il faut dire au compilateur **où elles sont déclarées**.

Le rôle de `#include`

La directive `#include` sert à **insérer le contenu d'un fichier dans un autre fichier** avant la compilation.

Exemple :

```
#include <stdio.h>
```

Cela signifie :

> *« Inclure les déclarations des fonctions d'entrée/sortie standard »*

Sans cette ligne :

```c
printf("Bonjour");
```

le compilateur **ne sait pas ce qu'est** `printf`.

`#include <...>` ou `#include "..."` : quelle différence ?

3.1 `#include <stdio.h>`

- Utilisé pour les **bibliothèques standard**
- Le compilateur cherche le fichier dans ses dossiers système

Exemples courants :

```c
#include <stdio.h>   // printf, scanf
#include <stdlib.h>  // malloc, rand, exit
#include <math.h>    // sqrt, pow
#include <string.h>  // strlen, strcmp
```

3.2 `#include "monfichier.h"`

- Utilisé pour **vos propres fichiers**
- Le compilateur cherche d'abord dans le dossier du projet

Exemple :

```c
#include "utils.h"
```

Qu'est-ce qu'un fichier `.h` ?

Un fichier `.h` (header / en-tête) **déclare ce qui existe**, mais ne contient pas le code complet.

Il sert à :

- déclarer des **fonctions**
- déclarer des **constantes**
- déclarer des **types**

▸ Exemple : `math_utils.h`

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

int addition(int a, int b);
int soustraction(int a, int b);

#endif
```

➡️ Ici, on **annonce** que ces fonctions existent quelque part.

Qu'est-ce qu'un fichier `.c` ?

Le fichier `.c` **contient le code réel**, c'est-à-dire les **définitions des fonctions**.

▸ Exemple : `math_utils.c`

```c
#include "math_utils.h"

int addition(int a, int b) {
    return a + b;
}

int soustraction(int a, int b) {
    return a - b;
}
```

➡️ Le `.c` implémente ce qui a été annoncé dans le `.h`.

Pourquoi séparer `.h` et `.c` ?

▸ Bonne pratique professionnelle

Cette séparation permet :

- une meilleure **lisibilité**
- une **réutilisation** du code
- un **travail en équipe**
- une **compilation plus claire**

▸ **Règle simple à retenir**

> ***Dans le `.h` : ce que le programme peut utiliser Dans le `.c` : comment cela fonctionne***

Exemple complet avec `main.c`

▸ `main.c`

```c
#include <stdio.h>
#include "math_utils.h"

int main(void) {
    int resultat;

    resultat = addition(5, 3);
    printf("Addition : %d\n", resultat);

    resultat = soustraction(10, 4);
    printf("Soustraction : %d\n", resultat);

    return 0;
}
```

Le rôle du `#ifndef` / `#define` / `#endif`

Dans un `.h`, ces lignes évitent **l'inclusion multiple**.

```c
#ifndef MATH_UTILS_H
#define MATH_UTILS_H
...
#endif
```

Sans ça :

- un même fichier pourrait être inclus plusieurs fois
- cela provoquerait des **erreurs de compilation**

➡️ C'est une **protection obligatoire** dans tout fichier `.h`.

Compilation avec plusieurs fichiers

Commande classique avec GCC :

```
gcc main.c math_utils.c -o programme
```

Explication :

- `main.c` → programme principal
- `math_utils.c` → fonctions externes
- `-o programme` → nom de l'exécutable

⚠️ **On ne compile jamais les** `.h`, seulement les `.c`.

Erreurs fréquentes à éviter

- Mettre du code de fonction dans un `.h`
- Oublier d'inclure son `.h` dans le `.c`
- Déclarer une fonction différemment dans le `.h` et le `.c`
- Inclure un `.c` dans un autre `.c`
- Oublier les gardes `#ifndef`

À retenir (synthèse)

- `#include` insère du code avant compilation
- `< ... >` → bibliothèques standard
- `" ... "` → fichiers du projet
- `.h` → déclarations
- `.c` → implémentations
- Chaque `.c` est compilé séparément
- Le `.h` sert de **contrat**

C - #include - TP 💪

> *Jusqu'à présent, tes programmes en C tenaient dans un seul fichier. Dans un vrai projet informatique, ce n'est plus le cas : le code est **séparé en plusieurs fichiers** afin d'être plus clair, plus facile à maintenir et réutilisable. Dans ce TP, tu vas apprendre à **organiser un programme C en plusieurs fichiers** en créant ta propre petite bibliothèque de fonctions.*

Objectifs

À la fin de ce TP, tu devras être capable de :

- utiliser correctement la directive `#include`
- différencier le rôle d'un fichier `.h` et d'un fichier `.c`
- créer et utiliser une bibliothèque personnelle
- compiler un programme composé de plusieurs fichiers source

Structure du projet

Ton projet doit respecter **exactement** la structure suivante :

```
tp_bibliotheque/
│
├── main.c
├── utils.h
└── utils.c
```

Étape 1 — Création des fichiers

Dans le dossier `tp_bibliotheque` :

1. Crée un fichier `main.c`
2. Crée un fichier `utils.h`
3. Crée un fichier `utils.c`

Étape 2 — Fichier `utils.h`

Dans le fichier `utils.h` :

1. Protège le fichier contre les inclusions multiples.
2. Déclare les **prototypes** des fonctions suivantes :

- une fonction qui retourne le plus grand de deux entiers
- une fonction qui retourne le plus petit de deux entiers
- une fonction qui indique si un nombre est pair

⚠️ **Aucune fonction ne doit être écrite dans ce fichier**, uniquement des déclarations.

Étape 3 — Fichier `utils.c`

Dans le fichier `utils.c` :

1. Inclue le fichier `utils.h`
2. Écris le code complet des fonctions déclarées dans le `.h`
3. Chaque fonction doit retourner une valeur correcte

Étape 4 — Fichier `main.c`

Dans le fichier `main.c` :

1. Inclue la bibliothèque standard nécessaire à l'affichage
2. Inclue ton propre fichier `utils.h`
3. Appelle chaque fonction avec des valeurs de test
4. Affiche clairement les résultats à l'écran

Résultat attendu à l'exécution (exemple)

Le programme doit afficher des résultats similaires à :

```
Max de 4 et 9 : 9
Min de 4 et 9 : 4
8 est pair
7 est impair
```

(Les valeurs de test peuvent être différentes.)

Étape 5 — Compilation

Compile ton programme avec une seule commande :

```
gcc main.c utils.c -o programme
```

L'exécutable obtenu doit fonctionner sans erreur.

Contraintes importantes

- Il est interdit de mettre du code de fonction dans un fichier `.h`
- Il est interdit d'inclure un fichier `.c` dans un autre `.c`
- Les fichiers `.h` ne sont jamais compilés seuls
- Le programme doit compiler sans erreur

Travail à remettre

- Le dossier `tp_bibliotheque`
- Les trois fichiers `.c` et `.h`
- Le programme doit compiler et s'exécuter correctement

Bonus (facultatif)

- Ajouter une fonction supplémentaire dans la bibliothèque
- Réorganiser le projet avec des sous-dossiers
- Ajouter des commentaires clairs dans le code

Si tu le souhaites, je peux aussi te fournir :

- la **fiche prof avec la solution**
- une **version corrigée commentée ligne par ligne**
- une **grille de critères d'évaluation prête à imprimer**

C - Créateur de joueur de foot 🏆

> *Projet*

> *Dans cet exercice, tu vas programmer un **créateur de joueur** en langage C. Chaque joueur reçoit un **capital de points aléatoire**, réparti automatiquement selon son **poste** (gardien, défenseur, milieu ou attaquant). Tu devras ensuite permettre à l'utilisateur de **distribuer ces points** dans les différentes **caractéristiques du joueur**, puis calculer une **note globale** en fonction de ses performances et de son comportement.*

Objectifs pédagogiques

Ce projet te permettra de mettre en pratique :

- les **variables** (types de base, structures),
- les **opérations arithmétiques** et les **conditions** (`if` , `else`),
- les **boucles** (`for` , `while`) pour les saisies répétées,
- les **fonctions** pour organiser ton programme,
- et la notion de **pondération** (répartition de points selon des pourcentages).

Principe du programme

1. L'utilisateur crée un joueur (nom, âge, numéro, poste).
2. Le programme **tire aléatoirement** un **budget total de points**, compris entre **60 % et 80 %** de 1200 points. → Ce total est donc compris entre **720 et 960 points**.
3. Ce budget est ensuite **réparti automatiquement** entre les **4 groupes d'attributs** (Physiques, Techniques, Tactiques/Mentaux et Généraux) selon le **poste choisi**.
4. L'utilisateur **distribue manuellement** les points de chaque groupe dans 4 caractéristiques, la **dernière** étant calculée automatiquement pour atteindre exactement le budget du groupe.
5. Enfin, le joueur entre ses **statistiques de saison** (buts, passes, arrêts, cartons, hors-jeu) et le programme calcule une **note finale sur 100**.

Pondérations par poste

Ces pondérations servent **au début** du programme pour déterminer le nombre de points alloués à chaque groupe d'attributs.

POSTE	PHYSIQUES	TECHNIQUES	TACTIQUES	GÉNÉRAUX
1 – Gardien	20 %	15 %	25 %	40 %
2 – Défenseur	25 %	15 %	35 %	25 %
3 – Milieu	25 %	30 %	30 %	15 %
4 – Attaquant	20 %	40 %	25 %	15 %

Étape 1 — Informations générales

À demander :

- `nom` (texte, 30 caractères max)
- `age` (entier, entre 10 et 45)
- `numero` (entier, entre 1 et 99)
- `poste` (1 à 4)

Exemple d'affichage :

```
═══ Créateur de joueur ═══
Nom du joueur : Alex
Âge (10..45) : 23
Numéro de maillot : 9
Poste [1=Gardien, 2=Défenseur, 3=Milieu, 4=Attaquant] : 1
OK : Alex (#9), 23 ans, Poste = Gardien
```

Étape 2 — Tirage du budget total et répartition

Le programme tire un **budget total aléatoire** entre 720 et 960 points :

```
--- Budget total ---
Tirage aléatoire (60%..80% de 1200) → budget_total = 900 pts
```

Il répartit ensuite ce budget entre les 4 groupes selon le **poste** choisi. Exemple pour un **gardien** (20%, 15%, 25%, 40%) :

```
--- Répartition par pondérations ---
Physiques  : 20% → 180 pts
Techniques : 15% → 135 pts
Tactiques  : 25% → 225 pts
Généraux   : 40% → 360 pts
Vérification : 180 + 135 + 225 + 360 = 900 ✅
```

> ⚠️ *Aucun groupe ne peut dépasser **400 points** (car 4 caractéristiques × 100 max).*

Étape 3 — Répartition manuelle dans chaque groupe

L'utilisateur distribue les points du groupe dans **3 attributs**, la **4e est calculée automatiquement** pour compléter le budget exact.

S'il dépasse le budget ou si la 4e valeur est hors limites (0..100), il doit recommencer la saisie du groupe.

▸ Exemple 3A – Groupe *Physiques* (budget 180)

```
--- Attributs PHYSIQUES (budget: 180 pts) ---
vitesse (0..100) ? 70   [reste: 110]
endurance (0..100) ? 50 [reste: 60]
force (0..100) ? 30     [reste: 30]
⟶ agilite = 30
Valider ce groupe ? (Y/n) Y
```

▸ Exemple 3B – Groupe *Techniques* (budget 135)

```
--- Attributs TECHNIQUES (budget: 135 pts) ---
précision (0..100) ? 50  [reste: 85]
contrôle (0..100) ? 40   [reste: 45]
dribble (0..100) ? 35    [reste: 10]
⟶ tir = 10
Valider ce groupe ? (Y/n) Y
```

Même logique pour les deux autres groupes :

- **Tactiques/Mentaux** : vision du jeu, défense, attaque, fair-play
- **Généraux** : placement, pressing, passe longue, duel aérien

---

Étape 4 — Statistiques du joueur

L'utilisateur encode les statistiques de saison :

```
--- Statistiques ---
Buts ? 3
Passes décisives ? 5
Arrêts ? 18
Cartons jaunes ? 1
Cartons rouges ? 0
Hors-jeu ? 2
```

Étape 5 — Calculs et note finale

1. Calcul de la **moyenne** de chaque groupe (sur 100).
2. Calcul d'un **score pondéré** selon le poste (moyenne des groupes × pondération).
3. Calcul d'un **score statistique** (selon le poste) :
   - Gardien : `5 × arrets`
   - Défenseur : `8 × buts + 5 × passes`
   - Milieu : `5 × buts + 8 × passes`
   - Attaquant : `10 × buts + 4 × passes` → Score limité à 100.
4. Calcul d'un **malus fair-play** :

```
malus = 2 × cartons_jaunes + 8 × cartons_rouges + hors_jeu
(capé à 30)
```

5. Calcul de la **note finale** :

```
note = 0.7 × score_attributs + 0.3 × score_statistiques − malus
(bornée entre 0 et 100)
```

Exemple de calcul affiché :

```
--- Calculs ---
Score attributs (pondéré) = 64.2
Score statistiques = 90
Malus fair-play = 4
Note finale = 67.9 / 100
```

Étape 6 — Fiche récapitulative

Affichage final attendu :

```
═══════════════ FICHE JOUEUR ═══════════════
Nom: Alex  (#9)   Âge: 23   Poste: Gardien
Budget total tiré : 900 pts
Budgets par groupe: Phys=180  Tech=135  Tact=225  Gen=360
-- PHYSIQUES : vit=70 end=50 for=30 agi=30   (moy=45.0)
-- TECHNIQUES: pré=50 ctr=40 dri=35 tir=10   (moy=33.8)
-- TACTIQUES : vis=60 def=55 att=60 fair=50  (moy=56.3)
-- GENERAUX  : pla=90 prs=85 pas=95 dua=90   (moy=90.0)
Stats: buts=3, passes=5, arrêts=18, CJ=1, CR=0, HJ=2
─────────────────────────────────────────────
Attributs pondérés : 64.2
Score statistiques  : 90
Malus fair-play     : 4
NOTE GLOBALE        : 67.9 / 100
Verdict : Excellent gardien sur sa ligne ; technique à perfectionner.
═══════════════════════════════════════════════
```

Détails de programmation attendus

- Structure `Joueur` avec sous-structures pour chaque groupe d'attributs.
- Fonctions conseillées :

```c
int tirerBudgetTotal(void);
void repartirBudgetParPoste(int poste, int total, int* bPhys, int* bTech, int* bTact, int
void saisirGroupe(const char* nom, int budget, int* a, int* b, int* c, int* d);
double calculScoreAttributs(int poste, double phys, double tech, double tact, double gen
double calculScoreStats(int poste, int buts, int passes, int arrets);
int calculMalus(int cj, int cr, int hj);
double calculNoteFinale(double attr, double stats, int malus);
void afficherFiche(const Joueur* j);
```

Critères d'évaluation

ÉLÉMENT	POINTS
Saisie + validations (groupes et bornes)	5
Gestion du budget global et des pondérations	4
Calculs corrects (scores, note finale, malus)	6
Code structuré et commenté	3
Affichage clair et complet	2



PROGRAMMATION EN PYTHON

Python

> *Python est un langage de programmation interprété, polyvalent et convivial qui a gagné en popularité au fil des ans. Créé par Guido van Rossum et publié pour la première fois en 1991, Python est devenu l'un des langages les plus utilisés dans le monde de la programmation en raison de sa simplicité, de sa lisibilité et de sa grande communauté de développeurs.*

Les avantages de Python

Simplicité et lisibilité

L'une des principales caractéristiques de Python est sa syntaxe simple et concise, qui le rend facile à apprendre et à utiliser même pour les débutants. La lisibilité du code Python est également un avantage majeur, permettant aux développeurs de comprendre rapidement le fonctionnement d'un programme.

Polyvalence

Python est un langage polyvalent qui peut être utilisé dans une grande variété de domaines, notamment le développement web, l'analyse de données, l'intelligence artificielle, l'automatisation, la robotique et bien plus encore. Sa polyvalence en fait un outil indispensable pour de nombreux développeurs et entreprises.

Grande bibliothèque standard

Python dispose d'une vaste bibliothèque standard qui offre des fonctionnalités prêtes à l'emploi pour diverses tâches de programmation. Cela permet aux développeurs d'économiser du temps en utilisant des modules et des packages déjà développés plutôt que de réinventer la roue.

Communauté active

La communauté Python est l'une des plus grandes et des plus actives dans le monde de la programmation. Cette communauté dynamique contribue au développement continu de Python en créant de nouveaux packages, en fournissant des ressources d'apprentissage et en soutenant les nouveaux arrivants.

Qui utilise Python et pourquoi ?

Développeurs web

Python est largement utilisé dans le développement web pour la création de sites web, d'applications web et de services web. Des frameworks populaires comme Django et Flask permettent de développer rapidement des applications web robustes et évolutives.

Scientifiques des données

Python est devenu le langage de choix pour de nombreux scientifiques des données en raison de ses puissantes bibliothèques d'analyse de données telles que NumPy, Pandas et Matplotlib. Ces outils facilitent l'exploration, la manipulation et la visualisation des données.

Ingénieurs en intelligence artificielle et en machine learning

Python est largement utilisé dans le domaine de l'intelligence artificielle et de l'apprentissage automatique en raison de sa facilité d'utilisation et de ses bibliothèques spécialisées telles que TensorFlow, Keras et scikit-learn. Ces bibliothèques permettent de développer et de déployer des modèles d'apprentissage automatique de manière efficace.

Développeurs de logiciels et d'applications

De nombreuses entreprises utilisent Python pour le développement de logiciels et d'applications en raison de sa polyvalence, de sa productivité et de sa capacité à s'intégrer facilement avec d'autres langages et technologies.

La popularité croissante de Python

Python connaît une popularité croissante grâce à ses nombreux avantages et à sa polyvalence. Selon divers indices de popularité, tels que l'indice TIOBE et l'enquête Stack Overflow Developer Survey, Python figure parmi les langages de programmation les plus populaires et les plus demandés dans l'industrie.

Sa popularité continue de croître en raison de son adoption par de grandes entreprises comme Google, Facebook, Amazon et Netflix, qui utilisent Python pour diverses applications et services.

En conclusion, Python est bien plus qu'un simple langage de programmation. C'est un outil puissant et polyvalent qui continue de gagner en popularité grâce à sa simplicité, sa lisibilité, sa grande communauté et sa polyvalence. Que vous soyez un débutant en programmation ou un développeur expérimenté, Python offre une multitude d'opportunités pour créer des applications innovantes et résoudre des problèmes complexes dans divers domaines.

Notes de cours

- Télécharger les notes de cours "officielles"

EXERCICES DU JOUR

- Exercices sur les variables
- Pour les courageux
- Exercices sur les conditions
- Exercices sur les boucles

Vidéos "Python au lycée"

- Premiers pas: console, calculs, fichiers .py et print (6')
- Premiers pas: variables, commentaires (8'30)
- Les 4 types de variables de base
- Si...alors...(sinon) (partie 1) structure et condition (8'50)
- Entrée au clavier: input(), int() et float() (9'44)
- Si...alors...(sinon) (partie 2) (5'20)
- Booléens, comparaisons, random() (8')
- Nombres au hasard ("aléatoires") (5'40)
- Les fonctions (Partie 1) (5'38)
- Les fonctions (Partie 2) (5'38)
- Les boucles **for** (5'38)
- Les boucles **for... in range()** (5'38)

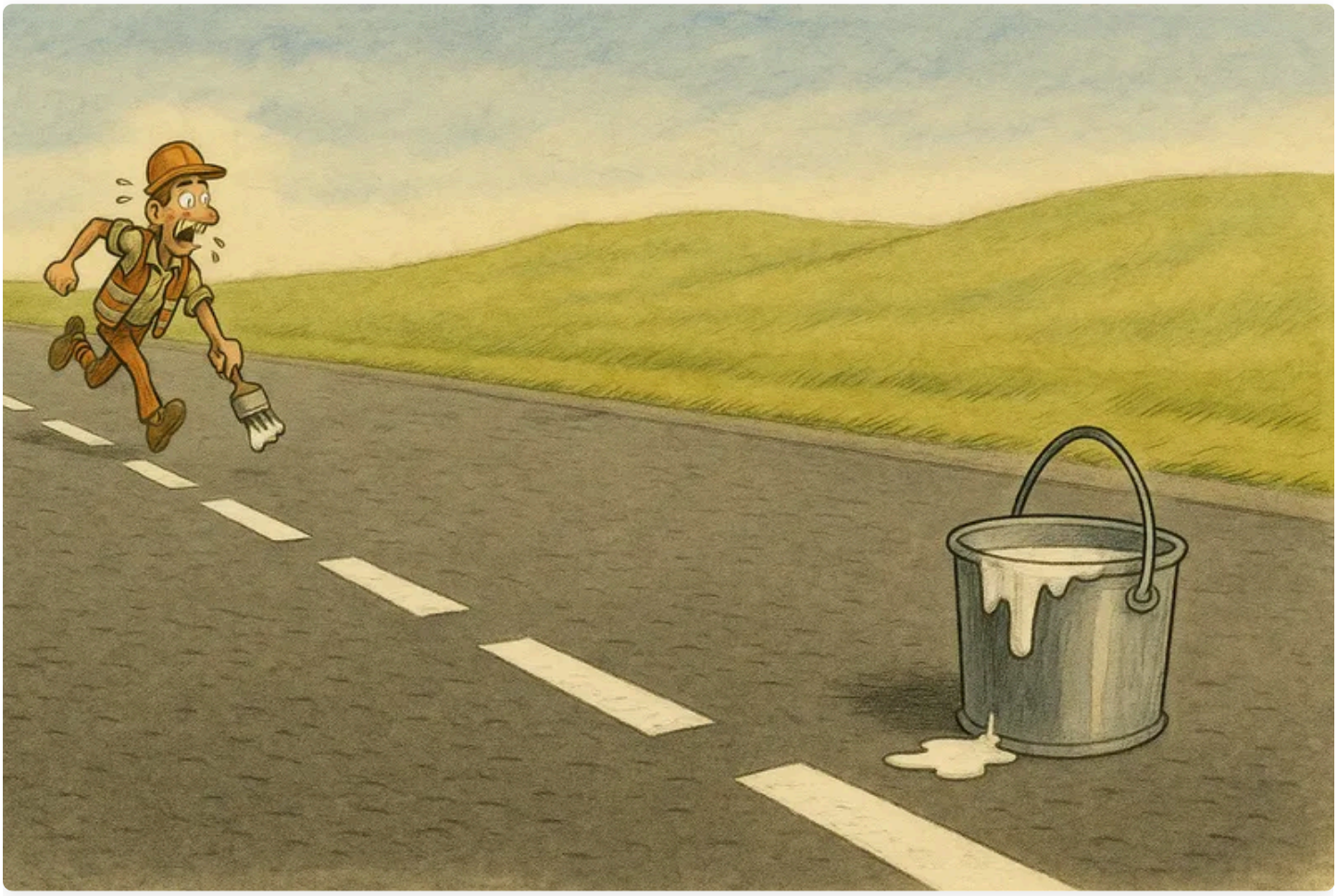
Tester ses programmes en Python

> *Quand tu écris un programme, tu veux être sûr qu'il **fonctionne correctement**. Un test, c'est un petit bout de code qui **vérifie automatiquement** qu'une partie de ton programme fait ce que tu attends.*

Faire des tests, ce n'est pas du temps perdu : c'est une **sécurité** qui permet de détecter des erreurs rapidement et d'être confiant quand tu modifies ton code.

Ecrire des tests, ça prend du temps. Mais ce temps, ce coût, est gagné et récupéré très rapidement par la suite. Il s'agit d'un **investissement**.

Ecrire du code et le modifier sans avoir de tests automatiques, c'est un comme si tu travaillais comme cet ouvrier qui peint une ligne blanche sur la route et revient toujours remplir son pinceau au pot de peinture...



Pourquoi écrire des tests ?

- **Détecter tôt les erreurs** : mieux vaut que ton test crashe en 2 secondes que de découvrir le bug pendant un projet complexe.
- **Répétables** : un test peut être relancé autant de fois que nécessaire, toujours sur le même jeu de données.
- **Documenter** : un test montre clairement ce que ton code est censé produire.

- **Éviter les régressions** : si tu améliores ton code, tes anciens tests garantissent que tu n'as rien cassé.

Principe : données maîtrisées → résultats attendus

L'idée est simple :

1. Tu choisis un **jeu de données d'entrée** simple et bien connu.
2. Tu décides du **résultat attendu**.
3. Tu écris un test qui exécute ta fonction et **compare** le résultat obtenu au résultat attendu.

▸ Exemple

Une fonction qui calcule la moyenne d'une liste de nombres :

```
def moyenne(notes):
    return sum(notes) / len(notes)
```

👉 Testons-la avec des données maîtrisées :

```
def test_moyenne():
    notes = [10, 20, 30]
    attendu = 20
    resultat = moyenne(notes)
    assert resultat == attendu, f"Attendu {attendu}, obtenu {resultat}"
```

- `assert` vérifie la condition : si c'est faux, le test échoue.
- Ici, avec `[10, 20, 30]`, on **sait** que la moyenne est `20`.

Organiser des tests en Python

▸ Option 1 — Tests manuels avec `assert`

Rapide à écrire, parfait pour débuter.

En Python, l'instruction `assert` sert à vérifier qu'une condition est vraie pendant l'exécution du programme. Si la condition est fausse, Python déclenche une erreur (`AssertionError`) et interrompt le programme. C'est un moyen simple de créer des **tests rapides**:

```
assert 2 + 2 == 4 # ok
assert 0 + 5 == 5 # ok
assert 0 * 5 == 5 # provoque une erreur
```

On peut donc aussi **comparer le résultat d'une fonction** avec la valeur attendue. Exemple :

```python
def addition(a, b):
    return a + b

assert addition(2, 3) == 5   # Passe
assert addition(0, 7) == 7   # Passe
assert addition(2, 2) == 5   # Échoue → AssertionError
```

En pratique, les tests sont regroupés dans une méthode *de test*:

```python
def test_addition():
    assert addition(2, 3) == 5   # Passe
    assert addition(0, 7) == 7   # Passe
    assert addition(2, 2) == 5   # Échoue → AssertionError
```

On peut aussi **ajouter un message** qui sera affiché en cas d'erreur:

```python
assert 2 == 3, "Devrait être égal"
```

Résultat:

```
Traceback (most recent call last):
  File "<string>", line 1, in <module>
AssertionError: Devrait être égal
```

👉 L'avantage est la **simplicité** : une seule ligne suffit pour vérifier. Mais l'inconvénient est que les messages d'erreur sont basiques, et on ne peut pas organiser facilement de gros ensembles de tests. C'est pourquoi `assert` est idéal pour **démarrer et tester localement**, puis on passe à des frameworks comme `unittest` ou `pytest` pour des projets plus grands.

▸ Option 2 — Utiliser `pytest` (simple et puissant)

`pytest` étant un module externe, il faut l'installer d'abord:

```
pip install pytest
```

Puis écrire des fonctions `test_...` :

```python
def test_moyenne():
    assert moyenne([10, 20, 30]) == 20
    assert moyenne([5, 15]) == 10
```

Et lancer (`cmd`):

```
pytest
```

Tu découvriras rapidement qu'il est généralement plus propre de placer les méthodes de test dans un fichier séparé.

▸ Option 3 — Utiliser `unittest` (librairie standard, mais plus compliquée)

```
import unittest

class TestMath(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(2 + 2, 4)
        self.assertEqual(0 + 5, 5)

if __name__ == "__main__":
    unittest.main()
```

▸ Différence entre `pytest` et `unittest` ?

Bonne question 👍

En Python, tu as donc deux approches principales pour les tests :

`unittest` (librairie standard)

- **Inclus par défaut** avec Python → rien à installer.
- Fournit une **structure claire** : classes de test, méthodes, assertions (`self.assertEqual` , `self.assertTrue` , etc.).
- Standard utilisé depuis longtemps → beaucoup de documentation et d'exemples.
- Compatible avec les environnements scolaires/entreprise où on **n'installe pas de paquets externes**.

👉 **Avantage pédagogique** : apprendre les concepts de test *sans dépendance supplémentaire*.

`pytest` (librairie externe)

- Installation : `pip install pytest` .
- Beaucoup plus **simple à écrire** : pas besoin de classes, juste des fonctions `test_...` .
- Messages d'erreurs très lisibles (il affiche directement l'expression qui a échoué).
- Permet facilement des tests avancés : fixtures, paramétrisation, mocks.
- Devenu le standard de fait dans beaucoup de projets pros.

👉 **Avantage pratique** : **moins de code, plus lisible pour débuter et pour des projets réels**.

Faut-il apprendre les deux ?

- Pour des **élèves débutants** : commencer avec `assert` puis `pytest` → plus clair, rapide à mettre en œuvre.

- Pour des **cours “pro”** : montrer aussi `unittest` car il fait partie de la bibliothèque standard, et certains environnements pros (ou examens) exigent de savoir l’utiliser.
- Les deux peuvent **coexister** : `pytest` sait exécuter les tests écrits avec `unittest`.

✅ En résumé :

- **unittest** = robuste, standard, utile si environnement limité.
- **pytest** = moderne, lisible, préféré dans les projets actuels.
- Mieux vaut connaître un peu des deux, mais **commencer par pytest** rend la découverte plus agréable.

Bonnes pratiques

- Utiliser des **jeux de données petits et clairs** (pas besoin de 1000 valeurs pour tester une moyenne).
- Tester des **cas normaux** mais aussi des **cas limites** (ex : liste vide, liste avec un seul élément, valeurs négatives, etc.).
- Donner des **noms explicites** aux tests (`test_moyenne_vide` , `test_addition_positifs`).
- Relancer les tests **après chaque modification** : ils doivent toujours passer.

Exemple concret : tester un algorithme

Imaginons un algorithme qui trouve le maximum d’une liste :

```
def maximum(liste):
    return max(liste)
```

Tests :

```
def test_maximum():
    assert maximum([1, 2, 3]) == 3
    assert maximum([-5, -2, -10]) == -2
    assert maximum([42]) == 42
```

Ces cas simples et **répétables** montrent que la fonction marche dans des situations différentes (positifs, négatifs, liste d’un seul élément).

À retenir

- Les tests permettent de **vérifier automatiquement** ton code.
- Ils doivent utiliser des **jeux de données maîtrisés** pour que le résultat attendu soit clair.

- Tu peux commencer avec des `assert`, puis découvrir `unittest` ou `pytest` plus tard.
- Tester ton code, c'est un réflexe de **pro** qui t'évitera beaucoup de bugs.

---

pytest - bonnes pratiques

> *`pytest` est simple à utiliser, mais pour que tes tests restent clairs et utiles à long terme, il est important d'adopter de bonnes pratiques. Voici une liste adaptée pour débiuter.*

Bonnes pratiques générales

- **Nommer clairement les tests**
  - Les fichiers doivent commencer par `test_` (ex: `test_algo.py`).
  - Les fonctions de test aussi (ex: `def test_moyenne():`).
  - Un bon nom explique ce qui est vérifié (`test_palindrome_mot_pair`, `test_moyenne_vide`).
- **Un test = un comportement à vérifier**
  - Évite de tout mettre dans une seule fonction.
  - Mieux vaut plusieurs petits tests faciles à lire qu'un énorme test qui fait tout.
- **Tester les cas limites**
  - Pas seulement les cas "normaux".
  - Exemples : liste vide, valeur négative, mot en majuscule, etc.
- **Messages clairs**
  - `pytest` affiche déjà les différences attendu/obtenu.
  - Tu peux ajouter un message en option :

    ```
    assert addition(2, 2) == 5, "2+2 devrait être égal à 4"
    ```
- **Relancer souvent les tests**
  - Après chaque modification importante → éviter les surprises.
  - Dans un vrai projet, on configure même les tests pour qu'ils tournent automatiquement.

Où placer les fonctions de test ?

1. Dans le même fichier que le code

- Pratique pour les **petits exercices** ou les cours.
- Tout est au même endroit, facile à lancer.
- Exemple :

```python
def addition(a, b):
    return a + b

def test_addition():
    assert addition(2, 3) == 5
```

👉 Bien pour apprendre et pour des scripts courts.

---

2. Dans un fichier séparé

- Standard dans les **vrais projets** :
  - `mon_code.py` → contient les fonctions.
  - `test_mon_code.py` → contient les tests.
- Avantages :
  - Le code de production et les tests ne se mélangent pas.
  - Facile à exécuter tous les tests en une commande :

    ```
    pytest
    ```
- Exemple :

```
projet/
  mon_code.py
  test_mon_code.py
```

👉 À privilégier quand le projet grandit.

En résumé

- Pour les **exercices en classe** → mettre les tests dans le même fichier est plus simple.
- Pour des **projets plus gros ou collaboratifs** → séparer le code et les tests est une bonne habitude.

---

pytest - exemple complet

> 👍 *Voici un **exemple plus complet** pour bien montrer l'usage de `pytest` avec plusieurs fonctions et ce qui se passe si un test échoue.*

Exemple complet avec `pytest`

Fichier : `test_algo.py`

```
# Fonctions à tester
def addition(a, b):
    return a + b

def moyenne(liste):
    return sum(liste) / len(liste)

def est_palindrome(mot):
    mot = mot.lower()
    return mot == mot[::-1]


# Tests
def test_addition():
    assert addition(2, 3) == 5
    assert addition(-2, 5) == 3
    assert addition(0, 7) == 7

def test_moyenne():
    assert moyenne([10, 20, 30]) == 20
    assert moyenne([5, 15]) == 10
    assert moyenne([42]) == 42

def test_palindrome():
    assert est_palindrome("radar") == True
    assert est_palindrome("kayak") == True
    assert est_palindrome("python") == False
```

Lancer les tests

Dans ton terminal :

```
pytest test_algo.py
```

Cas 1 — Tous les tests passent

Résultat attendu :

```
==================== test session starts ====================
collected 3 items

test_algo.py ...                                    [100%]

===================== 3 passed in 0.01s =====================
```

Le `...` signifie que les 3 fonctions de test ont réussi.

Cas 2 — Un test échoue

Supposons que tu aies fait une erreur volontaire dans `moyenne` :

```
def moyenne(liste):
    return sum(liste)   # ❌ oublié de diviser par len(liste)
```

Quand tu relances :

```
==================== test session starts ====================
collected 3 items

test_algo.py ..F                                       [100%]

================================ FAILURES ================================
_____________________________ test_moyenne _____________________________

    def test_moyenne():
>       assert moyenne([10, 20, 30]) == 20
E       assert 60 == 20
E        +  where 60 = moyenne([10, 20, 30])

test_algo.py:16: AssertionError
================== 1 failed, 2 passed in 0.02s ==========================
```

👉 `pytest` te montre directement :

- Quelle fonction a échoué (`test_moyenne`).
- Quelle ligne (`assert moyenne([10, 20, 30]) == 20`).
- Ce qu'il a obtenu (`60`) vs ce qu'il attendait (`20`).

À retenir

- `pytest` lance tous les fichiers `test_*.py` automatiquement.
- Le retour est **très lisible** : un point = un test réussi, `F` = échec.
- Les messages d'erreur sont clairs et montrent la différence entre attendu et obtenu.

Tester ses programmes en Python - Exercices ✍️

> *Voici quelques exercices pour pratiquer les tests en Python.*

Chaque exercice propose une fonction à écrire **et** des tests à définir.

Exercice 1 — Addition simple

Écris une fonction `addition(a, b)` qui retourne la somme de deux nombres.

Tests à écrire

- `addition(2, 3)` doit donner `5`.
- `addition(0, 7)` doit donner `7`.
- `addition(-2, 5)` doit donner `3`.

Exercice 2 — Moyenne

Écris une fonction `moyenne(liste)` qui calcule la moyenne des nombres d'une liste.

Tests à écrire

- `[10, 20, 30]` → `20`.
- `[5, 15]` → `10`.
- `[42]` → `42`.

⚠️ Bonus : que faire si la liste est vide ? (tester et lever une erreur `ValueError`).

Exercice 3 — Recherche dans une liste

Écris une fonction `contient(liste, valeur)` qui retourne `True` si la valeur est dans la liste, `False` sinon.

Tests à écrire

- `[1, 2, 3], 2` → `True`.
- `[1, 2, 3], 5` → `False`.
- `[], 7` → `False`.

Exercice 4 — Maximum

Écris une fonction `maximum(liste)` qui retourne le plus grand nombre d'une liste.

Tests à écrire

- `[1, 5, 3]` → `5`.
- `[-5, -2, -10]` → `-2`.
- `[42]` → `42`.

Exercice 5 — Palindrome

Écris une fonction `est_palindrome(mot)` qui retourne `True` si un mot est un palindrome (se lit pareil dans les deux sens).

Tests à écrire

- `"radar"` → `True`.
- `"python"` → `False`.
- `"Kayak"` → `True` (si on ignore la casse).

Exercice 6 — Filtrer les nombres pairs

Écris une fonction `pairs(liste)` qui retourne une nouvelle liste contenant seulement les nombres pairs.

Tests à écrire

- `[1, 2, 3, 4]` → `[2, 4]`.
- `[5, 7, 9]` → `[]`.
- `[]` → `[]`.

Exercice 7 — Tri croissant

Écris une fonction `trier(liste)` qui retourne une liste triée en ordre croissant.

Tests à écrire

- `[3, 1, 2]` → `[1, 2, 3]`.
- `[5]` → `[5]`.
- `[]` → `[]`.

Bonnes pratiques pour les élèves

- Chaque **test doit être lisible** : `assert fonction(entree) == sortie_attendue`.
- Les **cas limites** (listes vides, nombres négatifs, un seul élément) sont aussi importants que les cas “classiques”.
- Réexécute les tests après chaque modification.

Python: Les boucles

> *Les boucles sont des structures de contrôle fondamentales en programmation, permettant de **répéter l'exécution d'un bloc de code** un certain nombre de fois ou tant qu'une condition spécifiée reste vraie. Elles sont largement utilisées dans tous les langages de programmation pour automatiser des tâches répétitives, traiter des ensembles de données et réaliser des opérations itératives sur des structures complexes.*

Les boucles sont largement utilisées dans tous les langages de programmation **pour automatiser des tâches répétitives** et **traiter des ensembles de données**.

Utilité des Boucles en Programmation

En programmation, tu as souvent besoin de répéter une action plusieurs fois. Sans boucle, tu dois copier-coller le même code plusieurs fois, ce qui rend la maintenance difficile.

Compter sans boucle

Pour afficher les nombres de 1 à 5, tu écris :

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

Chaque instruction est rédigée individuellement.

Compter avec une boucle

Utilise une boucle `for` pour automatiser la répétition :

```
for i in range(1, 6):
    print(i)
```

La boucle parcourt automatiquement les nombres de 1 à 5. Tu écris le code une seule fois, ce qui simplifie la modification et améliore la lisibilité.

Les avantages des boucles

1. **Gain de temps** : Moins de code à écrire.
2. **Facilité de modification** : Modifier la plage de répétition est simple.
3. **Lisibilité améliorée** : Le code est plus structuré et compréhensible.

4. **Réduction des erreurs** : Moins de répétitions manuelles réduit le risque d'erreurs.

Utilité des boucles

Répétition de tâches

Les boucles permettent d'exécuter un bloc de code plusieurs fois sans avoir à le répéter manuellement. Cela permet d'économiser du temps et des efforts en automatisant des tâches répétitives.

Traitement de données

Les boucles sont couramment utilisées pour parcourir des ensembles de données, tels que des listes, des tableaux ou des fichiers, et effectuer des opérations sur chaque élément de manière itérative. Cela permet de traiter efficacement de grandes quantités de données.

Les Types de Boucles en Programmation

Il existe principalement deux types de boucles en programmation :

1. Boucles `for`

Les boucles `for` sont utilisées pour parcourir une séquence de données, telle qu'une liste, un tuple ou un dictionnaire, et exécuter un bloc de code pour chaque élément de la séquence. Elles sont couramment utilisées lorsque le nombre d'itérations est connu à l'avance ou lorsqu'il est nécessaire de parcourir une séquence de données.

2. Boucles `while`

Les boucles `while` sont utilisées pour exécuter un bloc de code tant qu'une condition spécifiée reste vraie. Elles sont couramment utilisées lorsque le nombre d'itérations n'est pas connu à l'avance ou lorsqu'une condition spécifique doit être vérifiée avant chaque itération.

Utilisation des Boucles en Python

En Python, les boucles sont largement utilisées pour diverses tâches de programmation en raison de leur simplicité et de leur flexibilité. Python offre les boucles `for` et `while`, ainsi que des fonctions intégrées telles que `range()` pour faciliter la création de boucles complexes.

À retenir

- Une boucle permet de répéter des instructions de manière automatique.
- La syntaxe de base en Python pour une boucle `for` est `for variable in range(début, fin)`.
- Utilise les boucles pour rendre ton code plus efficace et facile à maintenir.

En conclusion, les boucles sont des outils essentiels en programmation, offrant un moyen efficace de **répéter des tâches**, de traiter des données et de contrôler le flux d'exécution d'un programme.

Python: Les boucles while

> *Les boucles `while` sont une des structures de contrôle de Python. Elles permettent d'exécuter un bloc de code **tant qu'une condition spécifiée est vraie**. Dans cet article, nous explorerons en détail la syntaxe des boucles `while` en Python, leur fonctionnement, leurs avantages et leurs inconvénients, ainsi que des exemples pratiques pour illustrer leur utilisation.*

Introduction aux boucles `while`

Les boucles `while` sont utilisées lorsque vous souhaitez répéter l'exécution d'un bloc de code **aussi longtemps qu'une condition spécifiée reste vraie**. Autrement dit, **les boucles `while` permettent d'exécuter un bloc de code tant qu'une condition spécifiée est vraie**.

Les boucles `for`, quant à elles, sont utilisées pour parcourir des listes d'éléments (comme une liste, une chaîne de caractères ou une plage de nombres) et répéter l'exécution d'un bloc de code un certain nombre de fois.

💡 **1 Itération** = **1** passage dans la boucle.

💡 En Python on utilisera de préférence:

- les boucles ´for´ quand on connait le nombre d'itérations à l'avance
- les boucles `while` si on ne connait pas le nombre d'itérations à l'avance

La syntaxe générale d'une boucle `while` en Python est la suivante :

```
while condition:
    # Bloc de code à exécuter tant que la condition est vraie
```

Dans cette structure, le bloc de code indenté sous la déclaration `while` est répété tant que la condition spécifiée reste **vraie** (`True`). La condition est évaluée **avant** chaque itération de la boucle. Si la condition devient fausse à un moment donné, l'exécution de la boucle `while` se termine à la fin du bloc de coade et le programme passe à l'instruction suivante après la boucle.

Exemples d'utilisation des boucles `while`

Voyons maintenant quelques exemples pratiques pour illustrer l'utilisation des boucles `while` en Python :

Exemple 1 : Comptage jusqu'à un certain nombre

```
# Comptage jusqu'à 5
count = 1
while count <= 5:
    print(count)
    count += 1
```

Dans cet exemple, la boucle `while` est utilisée pour compter jusqu'à 5 en affichant chaque nombre à l'écran. La variable `count` est initialisée à 1, puis la boucle continue d'itérer tant que `count` est inférieur ou égal à 5. À chaque itération, la valeur de `count` est augmentée de 1 à l'aide de l'opérateur `+=`.

L'erreur la plus courante avec une boucle `while` consiste à oublier de mettre à jour la variable qui est testée dans la condition d'entrée. Dans le cas d'une boucle comme celle ci-dessus qui permet de compter jusqu'à un certain nombre, il est fréquent d'oublier d'incrémenter la variable (ici: `count`). Résultat? Une **boucle infinie**, c'est-à-dire qui ne se termine jamais...

Exemple 2 : Demande de saisie utilisateur

```
# Demande de saisie utilisateur jusqu'à ce qu'un mot spécifique soit entré
password = ""
while password != "motdepasse":
    password = input("Entrez le mot de passe : ")
print("Accès autorisé !")
```

Dans cet exemple, la boucle `while` est utilisée pour demander à l'utilisateur de saisir un mot de passe jusqu'à ce qu'il entre le mot de passe correct, qui est "motdepasse" dans cet exemple. La boucle continue de demander à l'utilisateur de saisir le mot de passe tant que la valeur entrée ne correspond pas au mot de passe attendu.

Avantages et inconvénients des boucles `while`

Les boucles `while` offrent plusieurs avantages, mais elles peuvent également présenter certains inconvénients :

Avantages :

- Flexibilité : Les boucles `while` permettent une **répétition basée sur une condition**, ce qui les rend flexibles et polyvalentes.
- Convivialité : Elles sont souvent utilisées lorsque **le nombre d'itérations n'est pas connu** à l'avance ou lorsqu'une condition spécifique doit être vérifiée avant chaque itération.

Inconvénients :

- **Risque de boucle infinie** : Si la condition spécifiée dans une boucle `while` reste toujours vraie, la boucle continuera à s'exécuter indéfiniment, ce qui peut entraîner un gel de l'exécution du programme ou un épuisement des ressources système.
- Complexité potentielle : Les boucles `while` peuvent rendre le code plus difficile à comprendre si la condition et les étapes d'itération ne sont pas clairement définies.

Différence entre `while` et `for`

Les boucles `while` et `for` sont deux structures de contrôle couramment utilisées en programmation pour répéter une série d'instructions.

La boucle `while` exécute un bloc de code **tant qu'une condition donnée est vraie**. Elle est particulièrement utile lorsque **le nombre d'itérations n'est pas connu à l'avance** et dépend d'une condition qui peut changer à chaque itération. Par exemple, une boucle `while` est idéale pour lire les entrées utilisateur jusqu'à ce qu'une entrée valide soit fournie.

En revanche, la boucle `for` est utilisée pour **itérer sur une séquence** (comme une **liste**, un tuple ou une **chaîne de caractères**) ou sur une **plage de valeurs prédéfinie** avec `range()`. Elle est particulièrement adaptée lorsque **le nombre d'itérations est connu à l'avance** ou lorsque l'on doit **parcourir chaque élément d'une collection**. Par exemple, une boucle `for` est idéale pour parcourir tous les éléments d'une liste et effectuer une opération sur chacun d'eux.

En résumé, utilisez une boucle `while` lorsque vous ne savez pas combien de fois vous devez itérer et une boucle `for` lorsque le nombre d'itérations est déterminé ou que vous itérez sur une collection.

Conclusion

Dans cet article, nous avons exploré les boucles `while` en Python, leur syntaxe, leur fonctionnement, ainsi que leurs avantages et leurs inconvénients. Les boucles `while` sont des outils puissants pour contrôler le flux d'exécution dans un programme et sont largement utilisées dans une variété de situations.

En comprenant comment utiliser efficacement les boucles `while`, vous pouvez écrire du code plus lisible, plus maintenable et plus robuste dans vos projets Python. N'oubliez pas de pratiquer et d'expérimenter avec des exemples concrets pour maîtriser pleinement les concepts discutés dans cet article.

Python: Les boucles while (exercices)

> *Voici 5 exercices/exemples d'utilisation des boucles `while` pour vous permettre de mieux comprendre leur fonctionnement.*

1. Deviner un nombre

Écris un programme où l'utilisateur doit deviner un nombre entre 1 et 100. Le programme continue de demander des entrées tant que l'utilisateur ne devine pas le bon nombre.

```
import random

nombre_secret = random.randint(1, 100)
devine = None

while #....
    # ....
    # ....
    # ....
    # ....

print("Bravo ! Vous avez trouvé le nombre secret.")
```

2. Somme des entiers positifs

Demande à l'utilisateur de saisir des entiers positifs et calcule la somme de ces entiers. Le programme se termine lorsque l'utilisateur saisit un nombre négatif.

```
somme = 0
nombre = 0

while nombre >= 0:
    nombre = int(input("Entrez un entier positif (nombre négatif pour arrêter) : "))
    if nombre >= 0:
        somme += nombre

print(f"La somme des nombres est : {somme}")
```

3. Calculer le nombre de chiffres d'un entier

Écris un programme qui prend un entier donné par l'utilisateur et utilise une boucle `while` pour calculer combien de chiffres contient ce nombre.

```python
nombre = int(input("Entrez un nombre : "))
compteur = 0

while nombre != 0:
    nombre //= 10  # Division entière par 10
    compteur += 1

print(f"Le nombre contient {compteur} chiffres.")
```

4. Table de multiplication

Demande à l'utilisateur de saisir un nombre, puis affiche la table de multiplication de ce nombre jusqu'à 10 en utilisant une boucle `while`.

```python
nombre = int(input("Entrez un nombre pour afficher sa table de multiplication : "))
i = 1

while i <= 10:
    print(f"{nombre} x {i} = {nombre * i}")
    i += 1
```

5. Saisir un mot de passe

Écris un programme qui demande à l'utilisateur d'entrer un mot de passe. Tant que le mot de passe est incorrect, il continue de demander à l'utilisateur de le saisir à nouveau.

```python
mot_de_passe_correct = "1234"
mot_de_passe_saisi = ""

while mot_de_passe_saisi != mot_de_passe_correct:
    mot_de_passe_saisi = input("Entrez le mot de passe : ")

print("Accès autorisé.")
```

Conclusion

Ces exercices couvrent différents cas d'utilisation des boucles `while`, depuis des boucles simples de contrôle jusqu'à des cas d'utilisation plus interactifs. Ils permettront à tes élèves de bien comprendre la structure et le fonctionnement de ce type de boucle.

Python: Les boucles while - Validation

> *La validation de l'input utilisateur est essentielle pour s'assurer que les données saisies sont correctes et évitent les erreurs de programme. En programmation, une bonne pratique consiste à vérifier les entrées des utilisateurs avant de les utiliser dans le programme. Cela permet d'éviter les erreurs et de garantir que les données sont conformes aux attentes.*

Validation d'un Input Utilisateur en Python

En Python, il est courant d'utiliser une **boucle** `while` pour demander l'input de l'utilisateur et **vérifier qu'il est valide**. Si l'input n'est pas valide, on redemande à l'utilisateur de saisir une valeur jusqu'à ce qu'une valeur valide soit fournie. Voici un exemple de code qui illustre comment valider qu'un nombre se trouve entre 1 et 5 :

Il existe 2 façons principales pour valider l'input utilisateur à l'aide d'une boucle `while` :

- une boucle infinie dont on sort dès qu'on a validé l'entrée utilisateur
- une boucle qui s'exécute tant que l'utilisateur n'a pas fourni de valeur valide

Vous pouvez choisir la version qui vous convient le mieux même si je trouve que la version boucle infinie est moins "propre" et moins correcte d'un point de vue algorithmique. Pourquoi? Parce dans ce cas la condition qui permet de sortir de la boucle est "cachée" quelque part dans le bloc de code au lieu de se trouver à côté du while.

D'un autre côté, cette version a l'avantage de ne devoir demander qu'une seule fois l'entrée à l'utilisateur (`input`)... alors que la version 'propre' nécessite de le faire 2x. Inspectez les programmes ci-dessous pour bien vous en rendre compte.

▸ Avec une boucle infinie

```
def demander_texte():

    while True:
        texte = input("Veuillez entrer un texte non vide: ").strip()
        if texte:
            return texte
        else:
            print("L'entrée ne doit pas être vide. Veuillez réessayer.")

# Utilisation de la fonction
texte_valide = demander_texte()
print(f"Vous avez entré le texte: {texte_valide}")
```

Remaquez l'instruction `return` à la ligne 5: c'est elle qui met fin à l'exécution de la boucle. Elle a pour effet de renvoyer la valeur, et donc de mettre fin à l'exécution de la fonction et, par conséquent, de la boucle.

Voici une explication détaillée du code fourni :

DÉCOMPOSITION DU CODE

1. **Déclaration de la fonction** `demander_texte()` :
   - La fonction est définie avec le mot-clé `def` suivi du nom de la fonction `demander_texte` et des parenthèses `()`.
2. **Boucle infinie** `while True` :
   - `while True` crée une boucle infinie qui continuera à s'exécuter jusqu'à ce qu'une condition de sortie soit rencontrée. Ici, la boucle se termine lorsqu'un texte valide est saisi et retourné.
3. **Demande d'input à l'utilisateur**:
   - `texte = input("Veuillez entrer un texte non vide: ").strip()` :
     - `input("Veuillez entrer un texte non vide: ")` affiche le message et attend que l'utilisateur saisisse du texte.
     - `.strip()` est utilisé pour enlever les espaces blancs en début et en fin de la chaîne de caractères saisie. Cela garantit que les entrées composées uniquement d'espaces sont traitées comme vides.
4. **Condition pour vérifier si le texte est non vide**:
   - `if texte:` vérifie si la chaîne de caractères `texte` n'est pas vide.
     - En Python, une chaîne de caractères vide `""` est évaluée comme `False`, tandis qu'une chaîne non vide est évaluée comme `True`.
5. **Retour du texte valide**:
   - `return texte` **renvoie le texte valide saisi par l'utilisateur, mettant ainsi fin à l'exécution de la fonction**.

▸ Sans boucle infinie

Voici un code qui produit le même effet, mais sans utiliser de boucle infinie:

```
def demander_texte():
    texte = input("Veuillez entrer un texte non vide: ").strip()
    while texte == "":
        print("L'entrée ne doit pas être vide. Veuillez réessayer.")
        texte = input("Veuillez entrer un texte non vide: ").strip()
    return texte

# Utilisation de la fonction
texte_valide = demander_texte()
print(f"Vous avez entré le texte: {texte_valide}")
```

Comme indiqué plus haut, tu peux remarquer aux lignes 2 et 5 qu'il faut utiliser 2x l'instruction qui demande l'entrée à l'utilisateur.

Vérifier/valider des nombres

La validation de l'input utilisateur permet d'éviter les erreurs et de garantir que les données sont conformes aux attentes.

En Python, il est donc courant d'utiliser une boucle pour demander l'input de l'utilisateur et vérifier qu'il est valide. Si l'input n'est pas valide, on redemande à l'utilisateur de saisir une valeur jusqu'à ce qu'une valeur valide soit fournie. Voici un exemple de code qui illustre comment valider qu'un nombre se trouve entre 1 et 5 :

```
def demander_nombre():
    nombre = input("Veuillez entrer un nombre entre 1 et 5: ")
    while not (nombre.isdigit() and 1 <= int(nombre) <= 5):
        print("Entrée invalide. Le nombre doit être un entier entre 1 et 5.")
        nombre = input("Veuillez entrer un nombre entre 1 et 5: ")
    return int(nombre)

# Utilisation de la fonction
nombre_valide = demander_nombre()
print(f"Vous avez entré le nombre {nombre_valide}.")
```

Explication du Code

La ligne 3 est importante: on vérifie que la chaîne de caractère ne contient que des chiffres et, si c'est le cas, alors on la convertit en entier.

1. **Initialisation de l'input**: On demande à l'utilisateur de saisir un nombre une première fois.
2. **Condition de la boucle `while`**: La boucle continue tant que l'input n'est pas valide. La validité est vérifiée en s'assurant que :
   - L'input est un chiffre (`nombre.isdigit()` vérifie que tous les caractères de la chaîne sont des chiffres).
   - L'input converti en entier est compris entre 1 et 5.
3. **Message d'erreur et nouvelle saisie**: Si l'input est invalide, un message d'erreur est affiché et l'utilisateur est invité à saisir un nouveau nombre.
4. **Retour du nombre valide**: Lorsque l'input est valide, la boucle se termine et la fonction retourne le nombre converti en entier.

Bonnes Pratiques

1. **Messages d'erreur clairs**: Fournissez des messages d'erreur clairs et instructifs pour aider l'utilisateur à comprendre ce qu'il a fait de mal et comment corriger son erreur.
2. **Validation stricte**: Assurez-vous de valider toutes les entrées utilisateur pour éviter les erreurs et les comportements inattendus.
3. **Boucles de validation**: Utilisez des boucles pour redemander l'input jusqu'à ce qu'il soit correct, ce qui améliore l'expérience utilisateur en évitant les plantages du programme.

Conclusion

Valider les entrées utilisateur est une pratique fondamentale en programmation. Elle permet d'éviter de nombreux problèmes et d'améliorer la robustesse des applications. En utilisant des boucles et des exceptions, vous pouvez gérer efficacement les entrées incorrectes et guider les utilisateurs vers des saisies valides.

Python: Les boucles for imbriquées

> *Lorsqu'on programme, il arrive souvent qu'on doive traiter des données organisées en plusieurs dimensions. Par exemple, imaginez que vous deviez parcourir un tableau en deux dimensions (comme une grille ou un échiquier), générer une table de multiplication ou dessiner des motifs complexes comme une pyramide d'étoiles. Dans ces cas-là, une simple boucle ne suffit pas, car il faut répéter des actions sur plusieurs niveaux.*

C'est ici que les **boucles imbriquées** deviennent indispensables. Elles permettent d'effectuer des actions à l'intérieur d'autres actions répétées, comme parcourir chaque cellule d'un tableau ligne par ligne, ou manipuler des ensembles de données structurées. Les boucles imbriquées sont donc la solution parfaite pour répondre à ces besoins de traitement multiple et de manipulation de données complexes.

Autrement dit, dès que tu dois manipuler/traiter des données organisées en plusieurs dimensions ou à plusieurs niveaux, tu devras probablement utiliser plusieurs boucles imbriquées. Et tu utiliseras autant de boucles imbriquées qu'il y a de dimmensions/niveaux.

Dans cet article, nous allons explorer les boucles imbriquées en Python, comprendre leur utilité et illustrer leur fonctionnement avec des exemples concrets.

Introduction aux Boucles Imbriquées

Les boucles imbriquées se produisent lorsque vous placez une boucle à l'intérieur d'une autre. Cela vous permet de parcourir des données en deux dimensions/niveaux, ou de répéter un ensemble d'actions de manière itérative en fonction de plusieurs critères.

La structure d'une boucle imbriquée en Python est assez simple :

```python
for i in range(3):  # Boucle extérieure
    for j in range(3):  # Boucle intérieure
        print(f"i = {i}, j = {j}")
```

Dans cet exemple, la boucle extérieure s'exécute trois fois, et à chaque itération de celle-ci, la boucle intérieure est exécutée trois fois à son tour. Le résultat est une série d'affichages où `i` prend les valeurs 0, 1, 2, et pour chaque `i`, `j` prend également les valeurs 0, 1, 2.

Résultat :

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
i = 2, j = 0
i = 2, j = 1
i = 2, j = 2
```

Exercices sur les Boucles Imbriquées

Voyons maintenant quelques exemples concrets d'utilisation des boucles imbriquées.

Afficher un carré de caractères

Ecris un programme qui permet d'afficher un carré de caractères (en utilisant toujours le même caractère).

Exemple pour un carré de 5x5:

```
*****
*****
*****
*****
*****
```

Afficher un carré de nombres aléatoires

Ecris un programme qui affiche un carré de nombres aléatoires.

Exemple pour un carré de 5x5:

```
5,7,8,9,5
3,2,4,7,5
0,9,8,7,5
3,8,2,1,0
9,6,3,5,4
```

Carré de Nombres Consécutifs

Les boucles imbriquées sont également utiles pour manipuler des données sous forme de grille ou de tableau. Par exemple, nous pouvons afficher un carré de nombres consécutifs.

```python
taille = int(input("Entrez la taille du carré : "))
compteur = 1

for i in range(taille):  # Boucle pour les lignes
    for j in range(taille):  # Boucle pour les colonnes
        print(f"{compteur:4}", end=" ")  # Affichage du nombre
        compteur += 1  # Incrémentation du compteur
    print()  # Changement de ligne
```

Résultat pour une taille de 4 :

```
   1    2    3    4
   5    6    7    8
   9   10   11   12
  13   14   15   16
```

Explication :

- La boucle extérieure parcourt les lignes du carré.
- La boucle intérieure parcourt les colonnes et affiche un nombre qui est incrémenté à chaque itération.
- L'utilisation de `f"{compteur:4}"` permet d'aligner les nombres correctement pour un affichage propre.

Triangle d'étoiles.

Ici vous allez afficher un triangle d'étoiles.

** Exemple de sortie pour une hauteur de 5**:

```
*
**
***
****
*****
```

Table de Multiplication

Un excellent exemple pour illustrer les boucles imbriquées est la création d'une table de multiplication. Dans cet exercice, nous allons afficher une table de multiplication allant de 1 à 10.

```python
for i in range(1, 11):  # Boucle pour les lignes
    for j in range(1, 11):  # Boucle pour les colonnes
        print(f"{i * j:4}", end=" ")  # Calcul du produit
    print()  # Changement de ligne après chaque ligne de la table
```

Résultat :

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Explication :

- La boucle extérieure contrôle la ligne de la table de multiplication.
- La boucle intérieure parcourt les colonnes et affiche le produit de `i` et `j`.
- Le `end=" "` dans la fonction `print()` empêche le saut de ligne automatique à la fin de chaque appel et aligne les résultats sur une seule ligne.

Damier (Échiquier) de Caractères

Enfin, un dernier exemple pour manipuler des boucles imbriquées consiste à créer un damier de caractères. Alternons les caractères `X` et `O` dans un damier de 8x8.

```python
for i in range(8):  # Boucle pour les lignes
    for j in range(8):  # Boucle pour les colonnes
        if (i + j) % 2 == 0:
            print("X", end=" ")
        else:
            print("O", end=" ")
    print()  # Changement de ligne
```

Résultat :

```
X O X O X O X O
O X O X O X O X
X O X O X O X O
O X O X O X O X
X O X O X O X O
O X O X O X O X
X O X O X O X O
O X O X O X O X
```

Explication :

- La boucle extérieure parcourt les lignes et la boucle intérieure les colonnes.
- À chaque position `(i, j)`, si la somme des indices est paire, un `X` est affiché, sinon c'est un `O`.

▸ Damier (Échiquier) de Caractères - v2

Pour afficher un véritable damier sur la console, vous pouvez remplacer les ' `0` ' par des ' ▒ ' (`alt-176`) et les ' `X` ' par des ' ▓ ' (`alt-178`).



▸ Générateur d'adresses IP

Une adresse IP est composée de 4 nombres allant de 0 à 255, séparés par des points. Ex: `192.168.0.1` . Elle possèdent donc 4 'dimensions' ou niveaux. Crée un programme capable de générer toutes les combinaisons possibles, depuis `0.0.0.0` jusqu'à `255.255.255.255` .

Cela pourrait par exemple être utilisé pour scanner un réseau et détecter les équipements accessibles. Tu verras que ça prend du temp pour générer et afficher les 255^4 combinaisons possibles. Dans ce cadre, on essaie généralement de se limiter à des ***ranges*** plus restreints. On pourrait par exemple scanner toutes les adresses du range `192.168.*.*` ou encore `10.10.10.*` .

▸ Attaque par force brute

Une attaque par force brute est une méthode pour deviner un mot de passe ou une clé en essayant toutes les combinaisons possibles jusqu'à ce que la bonne soit trouvée. Bien que cette méthode puisse être lente et inefficace face à des mots de passe longs ou complexes, elle reste une technique basique utilisée par les attaquants.

L'objectif d'une attaque par force brute est simple : **tester chaque combinaison possible d'un ensemble de caractères** jusqu'à trouver la bonne. Plus le mot de passe est long et plus l'ensemble de caractères est grand, plus l'attaque prendra du temps.

Pour générer toutes les combinaisons de caractères d'un mot de passe, on peut utiliser des boucles imbriquées. Imaginons que nous souhaitions tester toutes les combinaisons possibles d'un mot de passe composé de 4 caractères. Si les caractères disponibles sont [a-z], nous pouvons générer toutes les combinaisons possibles avec 4 boucles imbriquées.

Indice:

```
import string
print(string.ascii_letters) # lettres de a-z en minuscules et majuscules (52
print(string.ascii_lowercase) # lettres de a-z en minuscules (26 possibilités)possibilités)
```

GÉNÉRER LES COMBINAISONS

Crée un algorithme à base de boucles imbriquées qui permet de générer toutes les combinaisons de lettres possibles (de `a` à `z` en minuscules).

CHALLENGE EN CONDITIONS "RÉELLES"

Les mots de passe sont généralement cryptés avant d'être stocker sur un ordinateur. Une forme de cryptage qui était couramment utilisée est le hash md5. Il s'agit d'une fonction appliquée à un texte qui génère une clé "unique", le ***hash***, à partir de ce texte. Ce cryptage est irréversible. C'est à dire qu'il est impossible de retrouver mathématiquement le texte original depuis le hash généré.

Voici un exemple de fonction pour calculer le hash mld5 d'un texte en Python:

```
import hashlib

def md5(texte):

    # Créer un objet MD5
    hash_md5 = hashlib.md5()

    # Mettre à jour l'objet MD5 avec le texte (encodé en bytes)
    hash_md5.update(texte.encode('utf-8'))

    # Récupérer la valeur du hash sous forme hexadécimale
    md5_hash = hash_md5.hexdigest()

    return md5_hash
```

Ta misison: tu as récupéré une série de mots de passes *hashés* sur un serveur. Retrouve les mots de passe originaux (4 lettres) qui correspondent aux hash suivants:

- `881cc4157ed641a365a86452f27ed745`
- `40ea57d3ee3c07bf1c102b466e1c3091`
- `8d777f385d3dfec8815d20f7496026dc`
- `a7c3c2aa70d99921f9fb23ac87382997`
- `02c425157ecd32f259548b33402ff6d3`

ALLER PLUS LOIN

Maintenant, allons plus loin:

- trouve un moyen de **mesurer le temps** que prend ton programme pour trouver le mot de passe (en millisecondes).
- **mesure** les temps d'exécution pour trouver les 5 mots de passe précédents.
- double le nombre de combinaisons possibles en utilisant des lettres minuscules et majuscules
- **mesure** les temps d'exécution pour trouver les 5 mots de passe précédents.
- teste avec ce hash: `3f85a60ff25237dc58f2ad63a1c75e78`
- maintenant, on refait la même chose avec **5** caractères (minuscules et majuscules). Trouves les mdp suivants:

    - `a85e7c8ba38d00af2f47681123ae4ec2`
    - `522a8fa409901e49937e4261e60d1713`
- ** Que remarques-tu au niveau du temps d'exécution, d'abord en ayant doublé le nombre de caractères et ensuite en ayant ajouté **1 seul** caractère?

Ces exemples sont limités à des mots de passe d'une longueur fixe. Dans la pratique, les mots de passe peuvent être plus longs et l'ensemble des caractères beaucoup plus large (lettres majuscules, minuscules, chiffres, symboles, etc.). Pour générer des mots de passe de longueur variable, il est préférable d'utiliser une approche récursive ou de gérer dynamiquement les boucles.

3. Conclusion

Les boucles imbriquées sont des outils puissants pour effectuer des opérations répétitives sur des structures à plusieurs dimensions, comme des tableaux, des grilles ou des formes géométriques. Elles permettent d'automatiser et de simplifier des tâches complexes qui seraient fastidieuses à coder ligne par ligne.

Il est important de bien comprendre la logique des boucles imbriquées pour les utiliser efficacement et éviter les erreurs. Plus vous pratiquez avec des exemples comme ceux-ci, plus vous serez à l'aise avec leur utilisation dans des contextes variés.

N'hésitez pas à essayer ces exemples, et à les modifier pour explorer davantage le potentiel des boucles imbriquées dans vos propres projets Python.

Python: Les boucles for

> *Quand un programme commence à devoir répéter une action plusieurs fois, copier-coller la même ligne devient vite absurde. La boucle `for` est la réponse propre à ce problème : elle décrit une fois ce qu'il faut faire, et Python s'occupe de le refaire autant de fois qu'il faut.*

*Tu vas voir ici **pourquoi** on a inventé les boucles `for`, **comment** elles se construisent, et tu termineras en écrivant ta première table de multiplication automatique.*

🎯 Objectifs

À la fin de ce cours, tu seras capable de :

1. Reconnaître les situations où une boucle `for` remplace utilement du copier-coller.
2. Écrire une boucle `for` qui exécute une action **un nombre précis de fois**.
3. Utiliser `range()` avec un, deux ou trois paramètres.
4. **Parcourir** une chaîne de caractères, lettre par lettre.
5. Résoudre seul un petit problème classique (table de multiplication) avec une boucle.

Partie 1 — Le problème : copier-coller, ça ne tient pas

Imagine qu'on te demande d'afficher 5 fois **le même message** — par exemple « Joyeux anniversaire, Léa ! » pour faire une carte un peu folle. Sans boucle, ça donne :

```
print("Joyeux anniversaire, Léa !")
print("Joyeux anniversaire, Léa !")
print("Joyeux anniversaire, Léa !")
print("Joyeux anniversaire, Léa !")
print("Joyeux anniversaire, Léa !")
```

Cinq lignes identiques. Ça marche, mais maintenant :

- **Et s'il en fallait 200 fois ?** Tu vas vraiment écrire 200 `print` ?
- **Et si on veut changer le texte** (mettre « Bon anniversaire » à la place) ? Il faudrait modifier les 200 lignes.
- **Et si on ne sait qu'au moment de l'exécution combien de fois répéter** (l'utilisateur entre le nombre) ? Tu ne peux même pas écrire les `print` à l'avance.

> 🧠 ***La répétition manuelle ne passe pas à l'échelle.*** *Dès qu'on commence à copier-coller plus de deux ou trois lignes presque identiques, c'est le signe qu'il faut sortir une boucle.*

La boucle `for` répond exactement à ça : tu décris **une seule fois** ce qu'il faut faire, et Python s'occupe de le refaire autant de fois que tu veux.

> 💡 ***Et si chaque ligne était différente ?*** *Par exemple, afficher 5 prénoms d'élèves :* `Alice`, `Bob`, `Charlie` *… Ça se résout aussi avec une boucle, mais on a besoin d'une* ***liste*** *pour stocker les prénoms. Les listes arrivent dans un cours suivant — pour l'instant, on se concentre sur le cas « même action, répétée N fois ».*

Partie 2 — Première boucle : répéter N fois

Pour démarrer simple, on combine `for` avec la fonction `range()` pour répéter l'action **un nombre précis de fois** :

```
for i in range(5):
    print("Joyeux anniversaire, Léa !")
```

Résultat à l'écran :

```
Joyeux anniversaire, Léa !
Joyeux anniversaire, Léa !
Joyeux anniversaire, Léa !
Joyeux anniversaire, Léa !
Joyeux anniversaire, Léa !
```

Cinq fois. C'est exactement le programme du début, mais en **deux lignes** au lieu de cinq. Et si tu remplaces `range(5)` par `range(200)`, tu obtiens 200 lignes — sans rien changer d'autre.

▸ Décortiquons cette boucle

```
for      i      in     range(5)     :          ← en-tête (« la promesse »)
└──┬──┘ └─┬─┘ └──┬──┘ └─────┬─────┘
 mot-clé  variable          séquence à parcourir

    print("Bonjour")                            ← corps (indenté de 4 espaces)
```

ÉLÉMENT	RÔLE
`for`	Mot-clé qui démarre la boucle
`i`	Variable qui prend une nouvelle valeur à chaque tour
`in`	Mot-clé qui sépare la variable de la séquence
`range(5)`	Génère la séquence `0, 1, 2, 3, 4`
`:`	Termine l'en-tête (obligatoire en Python)
Corps indenté	Le code à répéter à chaque tour

> ⚠️ ***L'indentation est obligatoire.*** *Le corps de la boucle DOIT être indenté (4 espaces, ou une tabulation, mais reste cohérent). Sans indentation, Python ne sait pas où finit la boucle.*

Partie 3 — Anatomie d'une exécution

Reprends ce code, mais cette fois on **utilise** la variable `i` :

```
for i in range(5):
    print("Tour numéro", i)
```

Résultat :

```
Tour numéro 0
Tour numéro 1
Tour numéro 2
Tour numéro 3
Tour numéro 4
```

À chaque tour, `i` prend une valeur différente :

TOUR	VALEUR DE `I`	LIGNE AFFICHÉE
1	0	`Tour numéro 0`
2	1	`Tour numéro 1`
3	2	`Tour numéro 2`
4	3	`Tour numéro 3`
5	4	`Tour numéro 4`

> 💡 ***Pourquoi ça commence à 0 ?*** *Parce que* `range(5)` *produit* `0, 1, 2, 3, 4` *— cinq valeurs en partant de 0. C'est une convention en programmation : on compte à partir de 0. On verra dans la partie 5 comment forcer un autre point de départ.*

Partie 4 — Parcourir une chaîne de caractères

Il existe une autre situation où une boucle `for` est très utile : quand on veut **traiter chaque élément** d'une suite, sans forcément savoir combien il y en a.

Une chaîne de caractères, c'est exactement ça : une suite de lettres.

```
mot = "Python"

for lettre in mot:
    print(lettre)
```

Résultat :

```
P
y
t
h
o
n
```

Ici, on n'a pas écrit `range(6)`. On a écrit `for lettre in mot`. Python sait **tout seul** parcourir la chaîne lettre par lettre, peu importe sa longueur. Si le mot fait 6 lettres, la boucle s'exécute 6 fois. S'il en fait 50, elle s'exécute 50 fois.

> 🧠 ***À retenir*** *:* `for` *ne sert pas seulement à compter. Sa vraie nature est de dire « pour chaque élément de cette suite, fais ceci ». Une chaîne, une liste, un fichier ouvert… tout ça se parcourt avec* `for` *.*

▸ Exemple — Lister les lettres d'un mot

Tu pourrais essayer ça :

```
mot = "ornithorynque"

for lettre in mot:
    print("Une lettre :", lettre)
```

Sans rien savoir d'autre, ton programme va afficher 13 lignes. Tu n'as pas eu à écrire « 13 » dans ton code — c'est Python qui s'adapte.

Partie 5 — La fonction `range()` en détail

Tu as vu `range(5)`. Mais `range()` peut prendre **un, deux ou trois** paramètres. Chaque forme est utile dans des contextes différents.

▸ Forme 1 — `range(fin)` : de 0 à fin − 1

```
for i in range(5):
    print(i)
```

→ `0, 1, 2, 3, 4`

▸ Forme 2 — `range(début, fin)` : de début à fin − 1

```
for i in range(1, 6):
    print(i)
```

→ `1, 2, 3, 4, 5`

> 💡 ***Très pratique*** *pour compter de 1 à 10 « comme un humain », sans commencer à 0.*

▸ Forme 3 — `range(début, fin, pas)` : avec un incrément personnalisé

```
for i in range(0, 20, 2):
    print(i)
```

→ `0, 2, 4, 6, 8, 10, 12, 14, 16, 18`

Le pas peut être **négatif** pour aller à reculons :

```
for i in range(10, 0, -1):
    print(i)
```

→ `10, 9, 8, 7, 6, 5, 4, 3, 2, 1` (parfait pour un compte à rebours)

▸ Récapitulatif

FORME	GÉNÈRE	EXEMPLE TYPIQUE
`range(n)`	`0, 1, …, n − 1`	Répéter n fois
`range(a, b)`	`a, a+1, …, b − 1`	Compter de 1 à 10
`range(a, b, pas)`	`a, a+pas, …` (jusqu'à < `b`)	Pairs, impairs, à rebours

> ⚠️ ***La valeur de fin n'est jamais incluse.*** `range(1, 11)` *va de 1 à 10, pas jusqu'à 11. Ça surprend au début, mais c'est ce qui permet d'écrire* `range(len(mot))` *sans erreur.*

🧪 Exercices

▸ Exercice guidé — La table de 7

Énoncé. Écris un programme qui affiche la **table de multiplication par 7**, de 7 × 1 jusqu'à 7 × 10, dans ce format :

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
...
7 x 10 = 70
```

Avant de coder, prends 30 secondes pour réfléchir. **Ne lis pas la suite tout de suite** — essaie de réfléchir à ces trois questions :

1. Combien de lignes vais-je afficher au total ?
2. Qu'est-ce qui **change** d'une ligne à l'autre ?
3. Qu'est-ce qui **reste pareil** ?

ÉTAPE 1 — IDENTIFIER LA SÉQUENCE

Dix lignes à afficher : pour `i` qui va de **1 à 10**. C'est exactement ce que produit `range(1, 11)` (rappel : la valeur de fin n'est pas incluse, donc 11 pour aller jusqu'à 10).

Premier brouillon, juste pour vérifier que la boucle tourne bien :

```
for i in range(1, 11):
    print(i)
```

Exécute. Tu dois voir afficher les nombres de 1 à 10. Si oui, **la boucle est bonne** — il ne reste plus qu'à changer ce qu'on affiche.

ÉTAPE 2 — CALCULER LE PRODUIT

À chaque tour, on veut afficher `7 × i`. En Python, le `×` s'écrit `*` :

```
for i in range(1, 11):
    print(7 * i)
```

Tu obtiens :

```
7
14
21
28
35
42
49
56
63
70
```

Les bons résultats sont là. Mais le format n'est pas celui demandé.

ÉTAPE 3 — COMPOSER LA LIGNE COMPLÈTE

On veut afficher `7 x 1 = 7`, `7 x 2 = 14`, etc. La fonction `print` accepte plusieurs arguments séparés par des virgules : elle les affiche tous séparés par un espace.

```
for i in range(1, 11):
    print(7, "x", i, "=", 7 * i)
```

Résultat :

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
7 x 10 = 70
```

C'est exactement ce qui était demandé.

ÉTAPE 4 (BONUS) — PLUS PROPRE AVEC UNE F-STRING

Quand la ligne à afficher devient longue, les `print(a, "x", b, ...)` deviennent illisibles. Python a une syntaxe plus élégante : les **f-strings** (chaînes formatées).

```
for i in range(1, 11):
    print(f"7 x {i} = {7 * i}")
```

Le préfixe `f` devant les guillemets active la magie : tout ce qui est entre `{ }` est évalué et inséré dans la chaîne. C'est exactement la même sortie qu'avant, mais beaucoup plus lisible.

> 🧠 ***Habitue-toi aux f-strings dès maintenant.*** *C'est la façon moderne d'afficher du texte mêlé à des variables en Python. Tu vas en croiser dans presque tous les exercices à partir d'ici.*

POUR ALLER PLUS LOIN

Quelques variantes à tester par toi-même :

- **Table choisie** : remplace `7` par une variable `n = 7` en début de programme. Tu pourras changer la table en modifiant une seule ligne.
- **Table interactive** : remplace `n = 7` par `n = int(input("Quelle table ? "))`. Le programme demande à l'utilisateur quelle table afficher.
- **Toutes les tables de 1 à 10** : imbrique deux boucles `for` (mais ça, c'est pour un cours suivant).

▸ Exercice 2 — Prédiction

Avant de lancer chacun de ces programmes, **prévois sur papier** ce qu'il va afficher. Puis vérifie.

```
for i in range(5):
    print(i)
```

```
for i in range(1, 5):
    print(i)
```

```
for i in range(1, 11, 2):
    print(i)
```

```
for i in range(10, 0, -1):
    print(i)
print("BOUM !")
```

▸ Exercice 3 — Initiales en chaîne

Écris un programme qui demande ton prénom à l'utilisateur, puis affiche chaque lettre en majuscule, **une par ligne**. Indice : la méthode `.upper()` sur une lettre la met en majuscule.

```
Ton prénom ? Ludovic
L
U
D
O
V
I
C
```

✍️ À retenir

- Une boucle `for` répète un bloc de code, **soit un nombre précis de fois**, **soit pour chaque élément d'une séquence**.
- La syntaxe générale est :

```
for variable in sequence:
    instructions
```

- `range(n)` produit `0, 1, …, n - 1`. `range(a, b)` part de `a`. `range(a, b, pas)` ajoute un incrément (positif ou négatif).
- La **valeur de fin n'est jamais incluse** : `range(1, 11)` s'arrête à 10.
- Une chaîne de caractères est **itérable** : `for lettre in mot` parcourt les lettres une par une.
- L'indentation du corps de la boucle est **obligatoire**.
- Les **f-strings** (`f"… {variable} …"`) sont la façon moderne d'insérer des variables dans du texte.

Suite

Tu sais maintenant *répéter*. La prochaine étape : faire **évoluer une variable** d'un tour à l'autre. Ça permet de **compter** (combien de voyelles dans un mot ?) et **cumuler** (quelle est la somme des notes ?). C'est ce qu'on découvre dans le prochain cours.

Python: Compter et cumuler des valeurs dans une boucle

> *Lorsqu'on utilise une boucle, on ne fait pas que répéter des instructions. Très souvent, on veut aussi **compter des éléments** ou **additionner des valeurs** au fil des répétitions. Cet article explique comment fonctionnent ces mécanismes, comment les coder correctement, et surtout comment **raisonner** pour éviter les erreurs.*

Pourquoi compter ou cumuler dans une boucle ?

Une boucle `for` exécute un bloc de code **plusieurs fois**. Il est donc naturel de vouloir :

- **compter** combien de fois quelque chose se produit (ex. : nombre de lettres, nombre de valeurs, nombre d'essais) ;
- **cumuler** des valeurs (ex. : somme de nombres, total de points, score final).

👉 Ces deux idées reposent sur **le même principe fondamental**.

Le principe clé : une variable qui évolue

Pour compter ou additionner, on utilise toujours une **variable qui change de valeur** à chaque itération.

Cette variable doit :

1. être **initialisée avant la boucle** ;
2. être **modifiée à chaque tour de boucle**.

Compter : augmenter de 1 à chaque fois

Exemple simple : compter jusqu'à 5

```
count = 0

for i in range(5):
    count = count + 1

print(count)
```

▸ Ce qui se passe réellement

- `count` commence à `0`
- À chaque passage dans la boucle, on ajoute `1`
- Après 5 passages, `count` vaut `5`

👉 La ligne importante est :

```
count = count + 1
```

On peut la lire comme :

> *« le nouveau `count` devient l'ancien `count` plus 1 »*

Cumuler : additionner une valeur qui change

▸ Exemple : somme des nombres de 1 à 5

```
total = 0

for i in range(1, 6):
    total = total + i

print(total)
```

▸ Interprétation

- `total` commence à `0`
- `i` prend successivement les valeurs `1`, `2`, `3`, `4`, `5`
- À chaque tour, on ajoute `i` au total

👉 Ici, la ligne clé est :

```
total = total + i
```

Cela signifie :

> *« j'ajoute la valeur actuelle de `i` au total existant »*

Différence entre compter et cumuler

OBJECTIF	VARIABLE MODIFIÉE	CE QU'ON AJOUTE
Compter	`count`	toujours `+1`
Cumuler	`total`	une valeur (`i` , un score, un prix, etc.)

Erreur fréquente à éviter ❌

▸ Réinitialiser la variable dans la boucle

```
for i in range(5):
    count = 0      # ❌ erreur
    count = count + 1
```

➡️ Ici, `count` est remis à `0` à chaque tour. Résultat final : `count` vaut toujours `1` .

👉 **L'initialisation doit toujours être faite avant la boucle.**

Astuce essentielle : raisonner sur papier 🖍️

Quand un programme devient difficile à suivre, **il faut le simuler**.

▸ Méthode recommandée

1. Trace une **ligne par variable**
2. Fais évoluer les valeurs **itération par itération**

▸ Exemple de simulation sur papier

Code :

```
total = 0

for i in range(1, 4):
    total = total + i
```

Tableau de suivi :

TOUR	I	TOTAL AVANT	TOTAL APRÈS
1	1	0	1
2	2	1	3
3	3	3	6

➡️ Résultat final : `6`

👉 Cette méthode permet de :

- comprendre ce que fait réellement le programme ;
- repérer immédiatement les erreurs de raisonnement ;
- expliquer son code à quelqu'un d'autre.

Cas très courant : compter sous condition

Exemple : compter le nombre de lettres `"a"` dans un mot.

```
mot = "banane"
count = 0

for lettre in mot:
    if lettre == "a":
        count = count + 1

print(count)
```

Ici :

- la boucle parcourt chaque lettre ;
- le compteur n'augmente **que si la condition est vraie**.

À retenir

- Pour **compter**, on ajoute toujours `1` à une variable compteur.
- Pour **cumuler**, on ajoute une valeur qui change à une variable total.
- La variable de comptage ou de cumul :
  - est initialisée **avant** la boucle ;
  - est modifiée **dans** la boucle.
- En cas de doute, **simuler sur papier** avec un tableau est la meilleure stratégie.
- Comprendre l'évolution des variables est plus important que mémoriser le code.

Si tu veux, je peux ensuite :

- proposer **une série d'exercices ciblés "compter / cumuler"**,
- écrire un **article jumeau** sur les erreurs fréquentes,
- ou relier cette notion à des **cas concrets de jeu ou de statistiques**.

Python: Les boucles for sur des listes

> *Une boucle `for` ne sert pas seulement à répéter N fois la même chose. Sa vraie force, c'est de **parcourir une collection** d'éléments — une liste de prénoms, de prix, de notes — et de traiter chacun à tour de rôle. C'est ce qu'on apprend ici.*

*Dans le cours précédent, on a vu `for i in range(N)` pour répéter N fois la même action. Mais quand chaque tour de boucle traite un élément **différent** — un prénom, un prix, un nom d'animal — il faut une **liste**. C'est l'outil de stockage le plus basique de Python, et il marche main dans la main avec `for`.*

🎯 Objectifs

À la fin de ce cours, tu seras capable de :

1. Créer une **liste** Python avec les crochets `[ ]`.
2. **Parcourir** une liste avec une boucle `for`.
3. Appliquer la **convention de nommage** : nom de la liste au pluriel, variable d'itération au singulier.
4. Faire varier le traitement dans la boucle (afficher, modifier, conditionner).
5. Reconnaître qu'une liste peut contenir n'importe quel type de valeurs.

Partie 1 — Le problème : un truc différent à chaque tour

Reprends l'idée du cours précédent. On voulait afficher des prénoms d'élèves :

```
print("Alice")
print("Bob")
print("Charlie")
print("Diane")
print("Eli")
```

Avec ce qu'on a vu jusqu'ici (`for + range`), on **ne sait pas faire** : `for i in range(5)` répèterait 5 fois la **même** ligne. Mais nous, on veut **5 lignes différentes**.

Il manque un endroit où **stocker les cinq prénoms à l'avance**. C'est exactement le rôle d'une **liste**.

Partie 2 — Créer une liste

Une liste se crée avec des **crochets** `[ ]` et des éléments séparés par des virgules :

```
prenoms = ["Alice", "Bob", "Charlie", "Diane", "Eli"]
```

Quelques détails à remarquer :

- Le nom de la variable est **au pluriel** : `prenoms`. C'est une convention universelle en Python (et dans la plupart des langages), parce que la variable contient **plusieurs valeurs**.
- Chaque élément est entre **guillemets** : on stocke des chaînes de caractères.
- L'ordre est conservé : `prenoms[0]` vaut `"Alice"`, `prenoms[1]` vaut `"Bob"`, etc.

> 💡 ***Vocabulaire*** *: un élément d'une liste s'appelle aussi un* ***item*** *(GB* ***item****). On dit « parcourir les items » ou « traiter chaque item ».*

Partie 3 — Parcourir une liste avec `for`

C'est ici que tout se met en place. Pour traiter chaque prénom de la liste à tour de rôle :

```
prenoms = ["Alice", "Bob", "Charlie", "Diane", "Eli"]

for prenom in prenoms:
    print(prenom)
```

Résultat :

```
Alice
Bob
Charlie
Diane
Eli
```

Exactement ce qu'on voulait — **et la boucle s'adapte automatiquement** au nombre d'éléments. Si tu ajoutes 195 prénoms à la liste, la boucle s'exécute 200 fois sans rien changer d'autre.

▸ La convention de nommage : pluriel / singulier

Regarde bien les deux noms de variables dans la boucle :

```
for prenom in prenoms:
#   └─────┘    └─────┘
#  singulier   pluriel
#  (un item)   (la collection)
```

C'est **la** convention Python pour les boucles `for` sur listes. Elle se lit en français naturel :

> *« Pour chaque **prénom** dans la liste des **prénoms**, fais… »*

Autres exemples :

LISTE (PLURIEL)	VARIABLE DE BOUCLE (SINGULIER)	LECTURE
`animaux`	`animal`	« Pour chaque animal dans animaux »
`prix`	`prix_unitaire` (ou `p`)	« Pour chaque prix dans prix »
`notes`	`note`	« Pour chaque note dans notes »
`eleves`	`eleve`	« Pour chaque élève dans élèves »
`voitures`	`voiture`	« Pour chaque voiture dans voitures »

> 🧠 ***Pourquoi cette convention ?*** *Un lecteur qui découvre ton code doit pouvoir comprendre en deux secondes la nature de chaque variable. `for animal in animaux` est instantanément clair. `for x in liste1` est obscur. Adopte le pluriel/singulier dès le début, c'est gratuit et ça change la lisibilité.*

Partie 4 — Que faire dans la boucle ?

À l'intérieur de la boucle, tu peux faire **tout ce que tu veux** avec l'élément courant. Voici les usages les plus fréquents.

4.1 — Afficher chaque élément

C'est l'exemple qu'on vient de voir :

```
animaux = ["Chat", "Lapin", "Capibara", "Dragon de Komodo"]

for animal in animaux:
    print(animal)
```

4.2 — Afficher avec une mise en forme

```
animaux = ["Chat", "Lapin", "Capibara"]

# On utilise une f-string pour entourer le nom de texte
for animal in animaux:
    print(f"🐾 {animal} a été ajouté au zoo.")
```

Résultat :

```
🐾 Chat a été ajouté au zoo.
🐾 Lapin a été ajouté au zoo.
🐾 Capibara a été ajouté au zoo.
```

4.3 — Transformer chaque élément

```python
prenoms = ["alice", "bob", "charlie"]

# La méthode .upper() met une chaîne en majuscules
for prenom in prenoms:
    print(prenom.upper())
```

Résultat :

```
ALICE
BOB
CHARLIE
```

4.4 — Conditionner le traitement

Tu peux combiner `for` avec un `if` pour ne traiter qu'une partie des éléments :

```python
notes = [12, 8, 17, 4, 15, 19, 11]

# On affiche uniquement les notes en échec (< 10)
for note in notes:
    if note < 10:
        print(f"⚠️ Note insuffisante : {note}/20")
```

Résultat :

```
⚠️ Note insuffisante : 8/20
⚠️ Note insuffisante : 4/20
```

> 💡 ***Tu remarques*** *? La boucle parcourt* ***toutes*** *les notes, mais l'* `if` *filtre celles qu'on veut afficher. C'est la combinaison la plus utilisée en pratique.*

Partie 5 — Une liste peut contenir n'importe quoi

Les listes ne sont pas limitées aux chaînes. Tu peux y mettre des nombres, des booléens, des choses mélangées.

5.1 — Liste de nombres

```
prix = [2.50, 4.99, 1.20, 8.75, 3.00]

# Affichage formaté
for p in prix:
    print(f"{p:.2f} €")
```

Résultat :

```
2.50 €
4.99 €
1.20 €
8.75 €
3.00 €
```

> ***Le `:.2f` dans la f-string*** *force 2 chiffres après la virgule. Très pratique pour les montants.*

5.2 — Liste de booléens

```
reponses = [True, False, True, True, False]

for reponse in reponses:
    if reponse:
        print("✓ Correct")
    else:
        print("✗ Faux")
```

5.3 — Liste hétérogène (à éviter en général)

Python autorise des listes avec des types mélangés :

```
truc = ["Alice", 17, True, 3.14]

for element in truc:
    print(element)
```

Ça fonctionne, mais **évite-le** : une liste qui mélange tout, c'est un signe qu'on aurait dû utiliser une autre structure (un dictionnaire, par exemple — on verra plus tard).

> 🧠 ***Bonne pratique*** *: une liste =* ***une collection d'éléments du même type****, qui représentent la même chose.*

🧪 Exercices

▸ Exercice guidé — Le panier de courses

Énoncé. Tu as la liste des courses à faire :

```
courses = ["pain", "lait", "œufs", "fromage", "tomates"]
```

Écris un programme qui affiche :

```
À acheter :
- pain
- lait
- œufs
- fromage
- tomates

Total : 5 articles
```

ÉTAPE 1 — IDENTIFIER CE QU'IL FAUT

Trois choses à faire :

1. Afficher une ligne d'en-tête (`À acheter :`).
2. Parcourir la liste et afficher chaque article précédé d'un `-` .
3. Afficher une ligne de total.

Le **point 2** est le cœur de la boucle.

ÉTAPE 2 — LA BOUCLE QUI AFFICHE LES ARTICLES

```
courses = ["pain", "lait", "œufs", "fromage", "tomates"]

for course in courses:
    print(f"- {course}")
```

Note le **pluriel/singulier** : `for course in courses` .

Résultat à ce stade :

```
- pain
- lait
- œufs
- fromage
- tomates
```

ÉTAPE 3 — AJOUTER L'EN-TÊTE ET LE TOTAL

L'en-tête se met **avant** la boucle, le total **après**. La fonction `len()` donne la taille d'une liste.

```
courses = ["pain", "lait", "œufs", "fromage", "tomates"]

print("À acheter :")

for course in courses:
    print(f"- {course}")

print(f"\nTotal : {len(courses)} articles")
```

Le `\n` au début de la dernière f-string ajoute une ligne vide avant le total.

Résultat final :

```
À acheter :
- pain
- lait
- œufs
- fromage
- tomates

Total : 5 articles
```

🧠 ***Le squelette à retenir*** *: en-tête → boucle → bilan. C'est un pattern qu'on retrouve partout (factures, bulletins, statistiques…).*

▸ Exercice 2 — Le zoo

Tu as la liste suivante :

```
animaux = ["Chat", "Lapin", "Capibara", "Lion", "Dragon de Komodo"]
```

Écris un programme qui affiche pour chaque animal :

```
🐾 Chat fait partie du zoo. (4 lettres)
🐾 Lapin fait partie du zoo. (5 lettres)
🐾 Capibara fait partie du zoo. (8 lettres)
🐾 Lion fait partie du zoo. (4 lettres)
🐾 Dragon de Komodo fait partie du zoo. (16 lettres)
```

Indice : `len(animal)` te donne le nombre de caractères du nom (espaces compris).

▸ Exercice 3 — Notes au-dessus de la moyenne

Pour la liste suivante :

```
notes = [11, 7, 15, 18, 4, 13, 9, 16]
```

Affiche **uniquement** les notes ≥ 10, dans ce format :

```
✓ 11/20
✓ 15/20
✓ 18/20
✓ 13/20
✓ 16/20
```

Indice : combine `for` et un `if`.

▸ Exercice 4 — Inventaire avec prix

À partir des listes suivantes (qu'on suppose alignées dans le même ordre — produit i ↔ prix i) :

```
produits = ["Pain", "Lait", "Œufs", "Fromage"]
prix     = [1.50, 1.20, 2.80, 4.99]
```

Écris un programme qui affiche :

```
- Pain     : 1.50 €
- Lait     : 1.20 €
- Œufs     : 2.80 €
- Fromage  : 4.99 €
```

Indice : ici on a deux listes à parcourir en parallèle. Utilise `range(len(produits))` et accède aux deux listes par leur index — ou découvre seul la fonction `zip()` si tu es curieux.

✍️ À retenir

- Une **liste** se crée avec `[ ]` : `prenoms = ["Alice", "Bob"]`.
- Une boucle `for` parcourt **tous les éléments**, un par un, dans l'ordre.
- **Convention de nommage** : liste au **pluriel** (`animaux`), variable d'itération au **singulier** (`animal`).
- À l'intérieur de la boucle, tu peux **afficher**, **transformer**, **conditionner** — tout ce que tu veux faire avec l'élément courant.
- La boucle s'adapte **automatiquement** à la taille de la liste : pas besoin d'écrire combien d'éléments.
- Une liste peut contenir n'importe quel type, mais **par bonne pratique** elle contient des éléments de même nature.
- `len(liste)` te donne le nombre d'éléments.

Suite

Tu sais maintenant parcourir une liste. Une application directe très fréquente : **afficher un menu de sélection à l'utilisateur** — par exemple les choix d'un jeu, les rayons d'un magasin, les options d'une appli. C'est l'objet du cours suivant, qui montre la progression *menu manuel* → *menu avec une liste* → *menu numéroté avec* `enumerate()`.

Python: Afficher un menu de sélection

> *Un menu dans une console, c'est l'application la plus fréquente du combo *liste + boucle for*. On part d'un menu écrit à la main, on le remplace par un menu généré depuis une liste, puis on lui ajoute des numéros automatiques avec `enumerate()`. Et à la fin, l'utilisateur peut vraiment choisir.*

*Presque toutes les applications console montrent un menu à un moment ou l'autre : un jeu (« Nouvelle partie / Charger / Quitter »), une appli de quiz (« Choisis ta catégorie »), un script utilitaire (« Que veux-tu faire ? »). Ce cours montre la **bonne façon** de l'écrire — et surtout, comment passer du « ça marche pour 3 options » au « ça marche pour 30 ».*

🎯 Objectifs

À la fin de ce cours, tu seras capable de :

1. Écrire un menu console avec un titre, des options et un choix utilisateur.
2. Reconnaître pourquoi le menu « écrit à la main » devient ingérable au-delà de 3-4 options.
3. Générer un menu **depuis une liste**, avec une boucle `for`.
4. Ajouter une **numérotation automatique** grâce à `enumerate()`.
5. Récupérer le choix de l'utilisateur et l'utiliser pour accéder à l'option sélectionnée.

Partie 1 — Version 1 : le menu écrit à la main

Imagine que tu codes un mini-zoo et que tu dois afficher la liste des animaux disponibles pour que l'utilisateur en choisisse un. Première approche, naïve, mais qui marche :

```
print("Choisis un animal :")
print("Chat")
print("Lapin")
print("Capibara")
```

Résultat :

```
Choisis un animal :
Chat
Lapin
Capibara
```

Ça marche pour 3 animaux. Mais regarde ce qui cloche déjà :

- **Pas de numéros** : comment l'utilisateur exprime son choix ? Il doit taper le mot exact ? Avec ou sans accent ?
- **Pas évolutif** : si on ajoute un animal, il faut ajouter une ligne `print`.
- **Couplage fort** : le menu est figé dans le code, on ne peut pas le générer depuis un fichier ou une saisie.

> 🧠 ***Le symptôme*** *: plusieurs* `print` *qui se ressemblent comme deux gouttes d'eau. C'est exactement le signal qui dit « remplace-moi par une boucle ! ».*

Partie 2 — Version 2 : le menu via une liste + `for`

Première amélioration : ranger les animaux dans une **liste** et les afficher avec une boucle.

```
# La liste des animaux du zoo
animaux = ["Chat", "Lapin", "Capibara"]

print("Choisis un animal :")

# La boucle affiche un animal par ligne
for animal in animaux:
    print(animal)
```

Résultat identique à la version 1 :

```
Choisis un animal :
Chat
Lapin
Capibara
```

À première vue, c'est plus long que la version 1. **Mais** :

- **Pour ajouter un animal**, il suffit de l'ajouter à la liste : `animaux = ["Chat", "Lapin", "Capibara", "Lion"]`. La boucle l'affichera tout seul.
- **Pour 30 animaux**, ce code fait toujours 5 lignes.
- **Si la liste vient d'ailleurs** (un fichier, une base de données, une saisie utilisateur), seul `animaux = …` change. Le menu reste le même.

> 💡 ***Convention*** *rappelée du cours précédent : la liste est au* ***pluriel*** *(`animaux`), la variable de boucle au* ***singulier*** *(`animal`). Tu peux lire le code à voix haute : « pour chaque animal dans animaux, affiche-le ».*

C'est mieux, mais l'utilisateur ne sait toujours pas quoi taper pour choisir. Il faut un **numéro** devant chaque option.

Partie 3 — Version 3 : numéroter avec `enumerate()`

Ce qu'on veut afficher maintenant :

```
Choisis un animal :
1. Chat
2. Lapin
3. Capibara
```

Pour avoir un numéro qui s'incrémente automatiquement à chaque tour, Python fournit la fonction `enumerate()`. Elle prend une liste et te donne, à chaque tour, **deux valeurs** : l'index et l'élément.

```
animaux = ["Chat", "Lapin", "Capibara"]

print("Choisis un animal :")

for (i, animal) in enumerate(animaux):
    print(f"{i}. {animal}")
```

Résultat :

```
Choisis un animal :
0. Chat
1. Lapin
2. Capibara
```

Tu remarques quelque chose : la numérotation commence à **0**. C'est parce que Python compte à partir de 0 (l'index d'un élément dans une liste). Pour un humain, on préfère commencer à 1.

▸ Décortiquons `enumerate()`

```
for (i, animal) in enumerate(animaux):
#    └──────┬──────┘    └──────┬──────┘
#  deux variables     enumerate produit
#  par tour           des paires (index, élément)
```

À chaque tour de boucle, `enumerate` donne une **paire** :

TOUR	`I`	`ANIMAL`
1	0	`"Chat"`
2	1	`"Lapin"`
3	2	`"Capibara"`

> *Sans `enumerate()` tu aurais dû gérer un compteur à la main : `i = 0; for animal in animaux: print(i, animal); i += 1`. C'est plus long, plus fragile, et personne ne fait ça en Python — `enumerate` est l'outil dédié.*

▸ Commencer à 1 au lieu de 0

Deux possibilités :

SOLUTION A — AJOUTER 1 DANS LE `PRINT`

```
for (i, animal) in enumerate(animaux):
    print(f"{i + 1}. {animal}")
```

SOLUTION B — LE PARAMÈTRE `START` DE `ENUMERATE()`

```
for (i, animal) in enumerate(animaux, start=1):
    print(f"{i}. {animal}")
```

Les deux donnent le même résultat. La solution B est légèrement plus propre — `start=1` exprime clairement l'intention.

```
Choisis un animal :
1. Chat
2. Lapin
3. Capibara
```

Partie 4 — Récupérer le choix de l'utilisateur

Le menu est joli, mais à quoi sert-il si l'utilisateur ne peut pas répondre ? Ajoutons la **saisie**.

```
animaux = ["Chat", "Lapin", "Capibara"]

print("Choisis un animal :")

for (i, animal) in enumerate(animaux, start=1):
    print(f"{i}. {animal}")

# Récupère la réponse, convertit en entier
choix = int(input("Ton choix ? "))

# Retrouve l'animal correspondant (attention : -1 car les listes commencent à 0)
animal_choisi = animaux[choix - 1]

print(f"Tu as choisi : {animal_choisi}")
```

Déroulement à l'exécution :

```
Choisis un animal :
1. Chat
2. Lapin
3. Capibara
Ton choix ? 2
Tu as choisi : Lapin
```

> ⚠️ ***Pourquoi*** `choix - 1` ***?*** *L'utilisateur tape* `2` *pour le deuxième animal. Mais dans la liste, le deuxième animal est à l'index* `1` *(les listes commencent à 0). D'où le* `-1` *. Ne te trompe pas dans ce petit décalage, c'est la source d'erreur n°1 sur les menus.*

▸ Cas problématique : que se passe-t-il si l'utilisateur tape n'importe quoi ?

- Il tape `0` → `animaux[-1]` → renvoie le **dernier** animal (`Capibara`). Comportement bizarre mais pas plantage.
- Il tape `99` → `animaux[98]` → **plantage** (`IndexError`).
- Il tape `chien` → `int("chien")` → **plantage** (`ValueError`).

Pour gérer ces cas proprement, il faut ajouter une **validation** — c'est typiquement le rôle d'une boucle `while` qui redemande tant que la réponse est invalide. On voit ça dans le cours sur les boucles `while` avec validation.

Partie 5 — Le programme complet, commenté

Voici la version finale, avec commentaires pédagogiques :

```python
# --- Données du menu (modifie cette liste pour changer le menu) ---
animaux = ["Chat", "Lapin", "Capibara", "Lion", "Dragon de Komodo"]

# --- Affichage du menu ---
print("=== ZOO DU LYCÉE ===")
print("Choisis un animal à observer :")
print()  # ligne vide

for (i, animal) in enumerate(animaux, start=1):
    print(f"  {i}. {animal}")

print()

# --- Saisie du choix ---
choix = int(input("Ton choix ? "))

# --- Traitement du choix ---
animal_choisi = animaux[choix - 1]
print(f"\n→ Tu pars observer un {animal_choisi}. Bonne visite !")
```

Exécution :

```
=== ZOO DU LYCÉE ===
Choisis un animal à observer :

  1. Chat
  2. Lapin
  3. Capibara
  4. Lion
  5. Dragon de Komodo

Ton choix ? 3

→ Tu pars observer un Capibara. Bonne visite !
```

Note que pour **ajouter un animal**, il suffit de l'ajouter à la liste `animaux`. Tout le reste du programme s'adapte tout seul. **C'est ça, l'élégance d'un menu géré par boucle.**

🧪 Exercices

▸ Exercice guidé — Le menu du restaurant

Énoncé. Écris un programme qui :

1. Affiche le menu d'un restaurant avec ces plats : `["Lasagnes", "Salade César", "Burger", "Pizza margherita", "Tarte aux fruits"]`
2. Numérote chaque plat à partir de 1.
3. Demande à l'utilisateur son choix.
4. Affiche un message du type : `→ Commande validée : Pizza margherita.`

ÉTAPE 1 — RÉFLÉCHIR AU PLAN

Ce qu'il faut :

- Une **liste** des plats.
- Une **boucle for + enumerate** pour afficher avec numéros.
- Un **input** pour le choix.
- Un accès à la liste avec `[choix - 1]` pour récupérer le plat.

ÉTAPE 2 — LE SQUELETTE

```
plats = ["Lasagnes", "Salade César", "Burger", "Pizza margherita", "Tarte aux fruits"]

print("Menu du restaurant :")
# TODO : boucle d'affichage

# TODO : saisie du choix

# TODO : affichage de la commande
```

ÉTAPE 3 — LA BOUCLE D'AFFICHAGE

```
for (i, plat) in enumerate(plats, start=1):
    print(f"  {i}. {plat}")
```

(Convention pluriel/singulier : `plats` → `plat`.)

ÉTAPE 4 — LA SAISIE

```
choix = int(input("\nQuel plat veux-tu ? "))
```

ÉTAPE 5 — LA VALIDATION FINALE

```
plat_choisi = plats[choix - 1]
print(f"→ Commande validée : {plat_choisi}.")
```

ÉTAPE 6 — LE PROGRAMME COMPLET

```
plats = ["Lasagnes", "Salade César", "Burger", "Pizza margherita", "Tarte aux fruits"]

print("Menu du restaurant :")

for (i, plat) in enumerate(plats, start=1):
    print(f"  {i}. {plat}")

choix = int(input("\nQuel plat veux-tu ? "))
plat_choisi = plats[choix - 1]

print(f"→ Commande validée : {plat_choisi}.")
```

Exécution :

```
Menu du restaurant :
  1. Lasagnes
  2. Salade César
  3. Burger
  4. Pizza margherita
  5. Tarte aux fruits

Quel plat veux-tu ? 4
→ Commande validée : Pizza margherita.
```

▸ Exercice 2 — Le menu d'un jeu

Écris un menu de jeu vidéo avec les options :

```
options = ["Nouvelle partie", "Charger une partie", "Options", "Crédits", "Quitter"]
```

Le menu doit ressembler à ça :

```
╔═══════════════════╗
║   MENU PRINCIPAL  ║
╚═══════════════════╝

  1. Nouvelle partie
  2. Charger une partie
  3. Options
  4. Crédits
  5. Quitter

Ton choix > 1
Tu as choisi : Nouvelle partie
```

▸ Exercice 3 — Menu de classes du lycée

À partir de la liste :

```
classes = ["3TTR", "4TTR", "5TTR", "6TTR", "5TQ", "6TQ", "3GMS", "4GMS", "5GMS", "6GMS"]
```

Écris un programme qui demande à l'élève sa classe et affiche le message :

```
Classes disponibles :
  1. 3TTR
  2. 4TTR
  ...
  10. 6GMS

Ta classe ? 5
→ Bienvenue en 5TTR !
```

▸ Exercice 4 — Bonus : éviter la duplication

Tu veux pouvoir afficher **plusieurs menus différents** dans ton programme (menu principal, sous-menu options, etc.) — sans copier-coller la logique d'affichage. Écris une fonction `afficher_menu(titre, options)` qui prend un titre et une liste, et fait l'affichage. Indice : c'est juste la mise en commun de tout ce qu'on a vu, dans un `def`. (Si tu n'as pas encore vu les fonctions, garde cet exercice pour plus tard.)

✍️ À retenir

- Un menu console se construit avec **trois briques** : une liste d'options, une boucle `for + enumerate`, un `input`.
- `enumerate(liste)` donne à chaque tour une paire `(index, élément)` — utile dès qu'il faut un numéro devant les éléments.
- `enumerate(liste, start=1)` commence la numérotation à 1 — plus naturel pour un humain.
- Pour récupérer l'option choisie : `options[choix - 1]` (attention au `-1` puisque les listes commencent à 0).
- Pour ajouter ou retirer une option, on **modifie la liste** — rien d'autre ne change.
- La gestion des erreurs de saisie (lettres au lieu de numéros, numéro hors limites) demande une boucle `while` avec validation — sujet du cours suivant.

Suite

Le menu fonctionne, mais il **plante** si l'utilisateur tape n'importe quoi. Pour gérer ça proprement — redemander tant que la saisie est invalide — il nous faut une autre famille de boucles : **les boucles** `while`. On y vient.

Python: afficher un labyrinthe

> *Lorsqu'on programme, il arrive souvent qu'on doive traiter des données organisées en plusieurs dimensions. Par exemple, imaginez que vous deviez parcourir un tableau en deux dimensions (comme une grille ou un échiquier), générer une table de multiplication ou dessiner des motifs complexes comme une pyramide d'étoiles. Dans ces cas-là, une simple boucle ne suffit pas, car il faut répéter des actions sur plusieurs niveaux.*

Exercice Python : Affichage d'un labyrinthe en ASCII

Objectif : Créer un programme Python qui affiche un labyrinthe dans la console en utilisant les caractères ASCII ▒ (176) pour le sol et ▓ (178) pour les murs. Le labyrinthe doit être représenté sous la forme d'une matrice (tableau à deux dimensions).

▸ Consignes :

1. Le labyrinthe sera représenté par une **matrice** (ou tableau à deux dimensions) où chaque élément est soit un mur (`1`), soit un sol (`0`).
2. Le programme doit parcourir cette matrice et afficher chaque élément à l'aide des caractères ASCII suivants :
   - ▒ pour représenter le sol (correspondant à `0` dans la matrice).
   - ▓ pour représenter un mur (correspondant à `1` dans la matrice).

Utilisez 2 caractères ASCII pour obtenir un affiche plus "carré".

3. Afficher le labyrinthe dans la console sous la forme d'un tableau en parcourant la matrice.

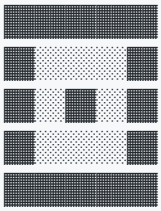
▸ Exemple d'un labyrinthe

Vous allez d'abord définir une matrice 5x5 de base qui ressemble à ceci :

```
1 1 1 1 1
1 0 0 0 1
1 0 1 0 1
1 0 0 0 1
1 1 1 1 1
```

Le programme doit alors afficher le labyrinthe dans la console en remplaçant les `0` par ▒ et les `1` par ▓.

Résultat attendu :



Étapes à suivre :

1. Définir une matrice contenant des `1` pour les murs et des `0` pour les espaces.
2. Créer une boucle qui parcourt chaque élément de la matrice.
3. Remplacer les `1` par `▓` et les `0` par `░` lors de l'affichage.
4. Afficher le résultat à la console.

Indications supplémentaires :

- Vous pouvez définir votre matrice comme suit en Python :

```
labyrinthe = [
    [1, 1, 1, 1, 1],
    [1, 0, 0, 0, 1],
    [1, 0, 1, 0, 1],
    [1, 0, 0, 0, 1],
    [1, 1, 1, 1, 1]
]
```

- Utilisez une boucle `for` pour parcourir les lignes de la matrice et une autre boucle `for` pour parcourir chaque élément de la ligne.

Étapes suivantes :

- Modifiez le labyrinthe pour qu'il soit de dimensions différentes.
- Utlisez Rich pour afficher le labyrinthe en couleurs
- Ajoutez un nouveau type de tuile qui représente des flaques d'eau ("▒" ascii-177)

Bonne chance !

Python: Les modules

> *Les modules en Python sont des fichiers contenant des définitions et des instructions. Ils permettent de diviser un programme en plusieurs fichiers pour une meilleure organisation et réutilisation du code. Un module peut définir des fonctions, des classes et des variables, ainsi que des blocs de code exécutables.*

Qu'est-ce qu'un module?

Définition

Un module est un fichier contenant du code Python avec l'extension `.py`. Il peut contenir des fonctions, des classes, des variables, et même d'autres modules.

Avantages

- **Réutilisation de code**: Facilite la réutilisation de code entre différents projets.
- **Organisation**: Permet une meilleure organisation du code en séparant les différentes parties logiques.
- **Espace de noms**: Évite les conflits entre noms de variables, fonctions, etc.

Utilisation des Modules Standards

Python est fourni avec une bibliothèque standard de modules. Voici comment les utiliser :

Importation

Pour utiliser un module, il faut d'abord l'importer avec le mot-clé `import`.

EXEMPLE:

```
import math
```

Accès aux Fonctions

Après avoir importé un module, tu peux accéder à ses fonctions et variables en utilisant la notation pointée.

EXEMPLE:

```
print(math.sqrt(16))  # Affiche 4.0
```

Création de tes Propres Modules

Créer un Fichier Module

1. **Écris du code**: Ouvre un éditeur de texte, écris ton code (par exemple, des fonctions ou des variables).
2. **Enregistre**: Sauvegarde le fichier avec un nom descriptif et l'extension `.py`.

EXEMPLE DE FICHIER `SALUTATIONS.PY`:

```
def bonjour(nom):
    return f"Bonjour, {nom}!"

def au_revoir(nom):
    return f"Au revoir, {nom}!"
```

Utilisation de ton Module

Après avoir créé ton module, tu peux l'importer et utiliser ses fonctions dans d'autres scripts Python.

EXEMPLE:

```
import salutations

print(salutations.bonjour("Alice"))  # Affiche "Bonjour, Alice!"
```

Importation Avancée

Importation Sélective

Si tu veux seulement importer certaines fonctions d'un module, tu peux le faire directement.

EXEMPLE:

```
from math import sqrt
print(sqrt(16))  # Affiche 4.0
```

Alias de Module

Tu peux aussi donner un alias aux modules importés pour raccourcir les noms.

EXEMPLE:

```
import numpy as np
```

Chemins de Module

Python recherche les modules dans les répertoires listés dans `sys.path`. Pour ajouter des répertoires à cette liste, tu peux modifier la variable d'environnement `PYTHONPATH` ou utiliser `sys.path.append(chemin)` dans ton script.

▸ **Exemple:**

```
import sys
sys.path.append('/chemin/vers/ton/module')
```

Bonnes Pratiques

- **Noms descriptifs**: Donne des noms clairs et descriptifs à tes modules et fonctions.
- **Documentation**: Documente tes modules et fonctions avec des docstrings.
- **Teste ton code**: Écris des tests pour vérifier que ton module fonctionne comme prévu.
- **Respecte PEP 8**: Suis les conventions de style de code de Python pour la lisibilité.

Conclusion

Les modules sont un outil puissant en Python qui te permettent de structurer ton code de manière logique, de réutiliser du code et de partager des fonctionnalités entre différents programmes. En apprenant à les utiliser et à les créer, tu pourras rendre tes projets Python plus organisés, efficaces et faciles à maintenir.

Exercice

Créez un module appelé `utils` avec les fonctions suivantes:

`celsius(temp)` : a function to convert temperatures from fahrenheit to Celsius. `pairs_impairs(liste)` : une fonction qui compte le nombre de nombres pairs et impaires dans une liste et affiche le résultat sur la console. `suite(debut, fin)` : une fonction qui renvoie (`return`) une liste de nombres compris entre `debut` et `fin` (inclus) qui sont divisibles par 7 et par 5.

Créez un fichier Python qui importe votre module et appelle/teste vos fonctions avec 3 cas différents à chaque fois.

ELECTRONIQUE

Les capteurs

> *Un capteur est un dispositif qui **détecte et mesure une grandeur physique ou une caractéristique de son environnement**, puis **convertit** cette information en un signal utilisable par un système électronique. Les capteurs jouent un rôle essentiel en robotique, car **ils permettent aux robots de percevoir** et d'interagir avec leur environnement.*

Utilité des capteurs

Les capteurs sont utilisés pour fournir aux robots des informations sur le monde qui les entoure. Ils permettent au robot de **recueillir des données sur des grandeurs physiques** telles que la lumière, le son, la température, la pression, la distance, etc. Ces informations sont ensuite utilisées **pour prendre des décisions**, effectuer des actions et interagir avec l'environnement de manière intelligente.

Les capteurs permettent aux robots de **détecter des objets**, d'analyser des caractéristiques de l'environnement, de suivre des lignes, d'éviter des obstacles, de mesurer des paramètres, de percevoir des mouvements, de reconnaître des formes et de nombreux autres aspects nécessaires à leur fonctionnement autonome.

Différents types de capteurs

On distingue généralement deux types principaux de capteurs en fonction de la manière dont ils fournissent des informations : les **capteurs analogiques** et les **capteurs numériques**.

- **Capteurs analogiques** : Les capteurs analogiques fournissent une sortie continue qui varie en fonction de la grandeur physique mesurée. Ils produisent **une tension ou un courant proportionnel à la grandeur mesurée**. Les valeurs de sortie varient généralement de manière linéaire ou non linéaire en fonction de la grandeur mesurée. Les signaux analogiques nécessitent souvent une conversion en signaux numériques pour être traités par un système électronique.
- **Capteurs numériques** : Les capteurs numériques fournissent une sortie discrète, généralement sous forme de bits (0 et 1) ou de valeurs numériques prédéfinies. Ils utilisent des composants électroniques internes pour convertir la grandeur physique en signaux numériques. Les capteurs numériques sont souvent plus précis, plus faciles à utiliser et peuvent être directement connectés à des microcontrôleurs ou à d'autres systèmes électroniques.

Il existe une grande variété de capteurs disponibles, chacun étant conçu pour mesurer une grandeur spécifique. Parmi les autres types de capteurs couramment utilisés en robotique, on peut citer les capteurs de mouvement, les capteurs de proximité, les capteurs de gaz, les capteurs de pression, les capteurs de température, les capteurs de lumière, les capteurs de force, les capteurs de couleur, les capteurs de son, les capteurs d'inclinaison, etc.

L'utilisation appropriée des capteurs permet aux robots de percevoir leur environnement, d'interagir de manière autonome et d'exécuter des tâches spécifiques en fonction des informations recueillies.

Principaux capteurs utilisés en robotique

Voici une liste des principaux capteurs utilisés en robotique, accompagnée d'une brève description de chacun :

Capteur de distance ultrasonique : Ce capteur utilise des ondes sonores pour mesurer la distance entre le robot et un objet environnant. Il est largement utilisé pour l'évitement d'obstacles et la navigation.

En classe, nous avons utilisé le capteur ultra-son HC-SR04, qui est un modèle très répandu et bon marché.

Capteur infrarouge : Les capteurs infrarouges détectent les rayonnements infrarouges émis par les objets. Ils sont utilisés pour la détection d'obstacles, le suivi de lignes, la détection de mouvement, etc.

Capteur de lumière : Les capteurs de lumière mesurent l'intensité de la lumière ambiante. Ils peuvent être utilisés pour détecter la luminosité de l'environnement et ajuster le comportement du robot en conséquence.

Capteur de température : Ce capteur mesure la température de l'environnement. Il est utilisé dans diverses applications telles que la surveillance de la température, la régulation thermique et le contrôle climatique.

En classe, nous avons utilisé le capteur de température et d'humidité DHT11, qui est un modèle très répandu et bon marché.

Capteur de pression : Les capteurs de pression mesurent la force appliquée sur une surface. Ils sont utilisés dans les applications de contrôle de la force, de détection de contact et de mesure de la pression atmosphérique.

Capteur de mouvement : Ces capteurs détectent les mouvements, les inclinaisons ou les rotations du robot. Ils sont couramment utilisés pour la détection de mouvement, la stabilisation et le contrôle de l'orientation.

Capteur de son : Les capteurs de son détectent les variations de pression acoustique dans l'environnement. Ils peuvent être utilisés pour détecter des sons spécifiques, effectuer une reconnaissance vocale ou analyser l'acoustique de l'environnement.

Capteur de couleur : Ces capteurs permettent de détecter les couleurs des objets. Ils sont utilisés dans les applications de tri, d'identification d'objets et de suivi de lignes basé sur la couleur.

Capteur de proximité : Les capteurs de proximité détectent la présence d'objets à proximité sans contact physique direct. Ils sont utilisés pour la détection d'obstacles, la détection de présence et l'activation de fonctions basées sur la proximité.

Capteur de gaz : Ces capteurs détectent la présence et la concentration de gaz dans l'environnement. Ils sont utilisés dans les applications de surveillance de la qualité de l'air, de détection de fuites de gaz, etc.

Oui, il existe d'autres capteurs utilisés en robotique. Voici quelques exemples supplémentaires :

Capteur d'inclinaison : Ces capteurs mesurent l'inclinaison ou l'orientation du robot par rapport à la gravité. Ils sont utilisés pour la stabilisation, la détection de l'attitude et le contrôle de l'équilibre.

Capteur de force : Les capteurs de force mesurent la force appliquée sur une surface ou un objet. Ils sont utilisés dans les applications de manipulation d'objets, de contrôle de la pression et de rétroaction haptique.

Capteur d'humidité : Ces capteurs mesurent le niveau d'humidité dans l'air ou dans un matériau. Ils sont utilisés dans les applications de contrôle de l'humidité, d'irrigation intelligente et de surveillance environnementale.

Capteur de gaz toxique : Ces capteurs spécifiques détectent la présence de gaz toxiques ou dangereux dans l'environnement. Ils sont utilisés dans les applications de sécurité industrielle, de surveillance environnementale et de détection de fuites.

Capteur de mouvement oculaire : Ces capteurs utilisent la technologie de suivi oculaire pour détecter et mesurer les mouvements des yeux. Ils sont utilisés dans les interfaces homme-machine avancées et les applications de réalité virtuelle.

Capteur d'odeur : Les capteurs d'odeur, également appelés capteurs de gaz odorants, peuvent détecter et analyser différentes odeurs. Ils sont utilisés dans des applications telles que l'analyse de la qualité de l'air, la détection d'odeurs spécifiques et le contrôle des émissions odorantes.

Capteur de vitesse : Ces capteurs mesurent la vitesse de déplacement du robot. Ils sont utilisés pour le contrôle de la vitesse, la détection de mouvement rapide et le suivi de la trajectoire.

Capteur de vision : Les capteurs de vision, tels que les caméras haute résolution, sont utilisés pour capturer des images et des vidéos de l'environnement du robot. Ils sont essentiels pour la perception visuelle, la détection d'objets, la reconnaissance faciale, etc.

Chaque capteur a des caractéristiques spécifiques et est utilisé pour des applications particulières en fonction des besoins du robot et de son environnement. L'utilisation de différents capteurs permet d'obtenir une perception complète et précise du monde extérieur, ce qui est essentiel pour des opérations robotiques avancées. Ces capteurs sont largement utilisés en robotique et offrent une multitude de possibilités pour la perception et l'interaction du robot avec son environnement. En fonction des besoins spécifiques de chaque projet, il est possible d'utiliser différents capteurs pour obtenir les informations nécessaires et permettre au robot d'interagir de manière intelligente avec le monde qui l'entoure.

BME280: pression atmosphérique et altitude

> *Dans cette activité, tu vas découvrir comment mesurer la pression atmosphérique avec le capteur BME280 et utiliser ces données pour calculer l'altitude, une compétence essentielle pour comprendre et analyser la descente d'un CanSat lors de son vol.*

🎯 Objectif du cours

Dans ce cours, tu vas apprendre à :

- lire la **pression atmosphérique** avec le capteur **BME280** via `raw_values` ,
- comprendre la relation entre pression et altitude,
- calculer l'**altitude** en utilisant une formule standard.

Ces compétences te serviront pour la **mission primaire** du CanSat.

Comprendre la pression atmosphérique

🌍 Qu'est-ce que la pression atmosphérique ?

La pression atmosphérique, c'est le **poids de l'air** au-dessus de toi.

- Au niveau de la mer : environ **1013 hPa**
- En altitude, la pression **diminue** car la colonne d'air est plus petite.

👉 Au CanSat, la pression va donc diminuer beaucoup *pendant la montée* puis augmenter pendant la descente.

Le capteur BME280

Mesures disponibles

- Température (°C)
- Pression (hPa)
- Humidité (%)

Communication

Il utilise le **bus I2C**, qui ne demande que deux fils : SDA et SCL.

▸ Exemple de branchement (Pico → BME280)

BME280	PICO
VCC	3.3V
GND	GND
SCL	GP9
SDA	GP8

Lire les données *numériques* avec `raw_values`

Le module BME280 utilisé dans le CanSat propose :

```
temp, pressure, humidity = bme.raw_values
```

Il retourne :

- la température en **°C**,
- la pression en **hPa**,
- l'humidité en **%**.

👉 Pas besoin de convertir des chaînes : c'est prêt pour les calculs.

▸ Exemple de code

```
from machine import I2C, Pin
from bme280 import BME280
import time

i2c = I2C(0, scl=Pin(9), sda=Pin(8))
bme = BME280(i2c=i2c)

while True:
    temp, pressure, humidity = bme.raw_values
    print("Température (°C):", temp)
    print("Pression (hPa):", pressure)
    print("Humidité (%):", humidity)
    print("-------------------------")
    time.sleep(1)
```

🔍 Comprendre le calcul de l'altitude à partir de la pression

Pour estimer l'altitude à partir de la pression mesurée par le BME280, on utilise une formule barométrique simplifiée. Mais avant de l'utiliser, il faut comprendre un point essentiel : **la pression au niveau de la mer n'est pas une constante !**

🌊 Qu'est-ce que la pression MSL ?

MSL signifie **Mean Sea Level pressure** = pression ramenée au niveau de la mer. C'est la valeur que les stations météo publient dans les bulletins.

IMPORTANT :

- La valeur "standard" est **1013.25 hPa**, mais elle varie **chaque jour**, selon la météo.
- Lors d'un **anticyclone**, la pression peut monter jusqu'à **1035 hPa**.
- Lors d'une **dépression**, elle peut descendre vers **980 hPa** (ou moins).

👉 Si tu utilises la valeur **standard 1013.25 hPa** alors que la pression réelle du jour est par exemple 1002 hPa, ton altitude sera fausse de plusieurs **dizaines de mètres**.

📡 Où trouver la pression MSL du jour ?

Voici des sources fiables (utiliser la station météo la plus proche) :

- **IRM – Belgique** https://www.meteo.be Chercher "pression atmosphérique" → valeur en hPa.
- **OpenWeatherMap** (API ou application) https://openweathermap.org
- **Météociel** https://www.meteociel.fr/observations-meteo/pression.php
- **Applications smartphone** : Météo & Radar, Windy, etc.

👉 Dans un projet CanSat, on choisit la station météo la plus proche du lieu du **lancement**.

🧮 La formule utilisée

La relation pression ↔ altitude vient de la physique de l'atmosphère :

$$\text{altitude} = 44330 \times \left(1 - \left(\frac{P}{P_0}\right)^{\frac{1}{5.255}}\right)$$

- **P** = pression mesurée par le BME280 (en hPa)
- **P_0** = pression MSL (pression au niveau de la mer, en hPa)
- **altitude** en mètres

POURQUOI ÇA FONCTIONNE ?

L'air est comprimé par son propre poids. Plus on monte :

- la densité de l'air diminue,

- la pression diminue.

La formule ci-dessus exprime précisément cette diminution.

▸ 📘 Exemple concret

Imaginons que tu mesures :

- Pression du capteur : **955 hPa**
- Pression MSL du jour (IRM) : **1002 hPa**

$$altitude = 44330 \times \left(1 - (955/1002)^{1/5.255}\right)$$

→ Résultat ≈ **423 m**

Si tu avais utilisé 1013.25 hPa par défaut :

→ ≈ **500 m** (erreur de +77 m)

▸ 🌡️ Pourquoi la pression MSL change ?

Parce que la pression dépend de la masse d'air au-dessus de la région :

- Air chaud → plus léger → **basse pression**
- Air froid → plus lourd → **haute pression**
- Mouvements atmosphériques → variations permanentes

Donc, **tu dois calibrer ton capteur** en mettant à jour le `P0` **chaque jour** ou avant chaque vol.

🌍 Faut-il utiliser la pression MSL du lieu ?

Oui, mais il faut bien comprendre ce que cela signifie.

▸ 🔹 1) La station météo te donne *la pression ramenée au niveau de la mer*

Quand tu vas sur IRM, OpenWeather, Windy, etc., la pression affichée n'est **pas** la pression réellement mesurée à cet endroit. Les stations appliquent un calcul pour "ramener" la pression mesurée au **niveau de la mer (MSL)** afin de pouvoir comparer deux stations situées à des altitudes différentes.

Exemple :

- Pression mesurée à Bastogne (500 m) ≈ 955 hPa
- Pression ramenée au niveau de la mer (MSL) ≈ 1010 hPa

👉 C'est cette **pression MSL** (1010 hPa) qu'il faut utiliser dans la formule d'altitude.

Pourquoi ? Parce que la formule barométrique **suppose que P_0 correspond à la pression au niveau de la mer**, pas à la pression "au sol" à 350 ou 500 m d'altitude.

🔹 Alors quelle valeur utiliser dans le CanSat ?

✓ Option 1 (recommandée par l'ESA & IRM)

➡️ **P_0 = pression MSL du jour fournie par la station météo la plus proche du site de lancement.**

Sources :

- IRM Belgique : https://www.meteo.be
- Windy : https://www.windy.com
- OpenWeatherMap
- Stations météo universitaires locales

C'est la méthode "officielle".

🔹 Option 2 : calibration locale pour plus de précision

Les conditions météo changent en permanence. Même la valeur MSL donnée par IRM peut entraîner une erreur (surtout si la station est loin).

👉 Solution : **calibrer P_0 sur place**.

ÉTAPES :

1. Tu regardes la **vraie altitude** du site (Google Maps, géoportail, panneau local).
2. Tu mesures la pression du BME280 sur place : `P_mesure`.
3. Tu ajustes P_0 pour que la formule te redonne cette altitude.

EXEMPLE

Lieu du lancement : altitude réelle = **510 m** Pression mesurée BME280 = **955.4 hPa**

Tu cherches la valeur P_0 qui donne 510 m :

[510 = 44330 \times \left(1 - (955.4/P_0)^{1/5.255}\right)]

Résultat → **P_0 ≈ 1010.6 hPa**

👉 Tu mets `P0 = 1010.6` dans ton code.

C'est la méthode la **plus précise** possible.

🔹 5) Mini-algorithme conseillé pour le CanSat

1. Récupérer la pression MSL du jour (IRM).
2. Mesurer la pression réelle au sol avec le BME280.
3. Calculer l'altitude affichée → comparer avec l'altitude connue.
4. **Ajuster P_0 automatiquement** dans le code pour corriger l'erreur.
5. Laisser P_0 constant pendant tout le vol.

Calculer l’altitude à partir de la pression

On utilise la formule barométrique simplifiée :

$$\text{altitude} = 44330 \times \left(1 - \left(\frac{P}{P_0}\right)^{\frac{1}{5.255}}\right)$$

- **P** : pression mesurée (hPa)
- **P_0** : pression au niveau de la mer (par défaut 1013.25 hPa)

Code "complet" avec altitude

```
from machine import I2C, Pin
from bme280 import BME280
import time

i2c = I2C(0, scl=Pin(9), sda=Pin(8))
bme = BME280(i2c=i2c)

P0 = 1013.25   # pression standard au niveau de la mer

while True:
    temp, P, hum = bme.raw_values

    # ICI: CALCULE L'ALTITUDE AVEC LA FORMULE SIMPLIFIEE

    print(f"Température: {temp:.2f} °C")
    print(f"Pression: {P:.2f} hPa")
    print(f"Altitude estimée: {altitude:.2f} m")
    print("------------------------------")

    time.sleep(1)
```

Exemple d'interprétation

PRESSION MESURÉE (HPA)	ALTITUDE ESTIMÉE
1013	0 m
1000	~110 m
950	~520 m
900	~980 m
800	~1900 m

👉 Pendant le vol du CanSat, tu vas observer une **chute rapide de pression**, ce qui te permet de **recalculer l'altitude en temps réel**.

À retenir

- Le BME280 renvoie des **valeurs numériques** grâce à `raw_values`.
- La pression diminue quand l'altitude augmente.
- On peut calculer l'altitude **en direct** pendant le vol.
- Ces données sont la base de la **mission primaire** du CanSat.

💡 Exercice

1. Lire les valeurs du capteur avec `raw_values`.
2. Afficher température + pression + altitude chaque seconde.
3. Monter un étage avec le capteur et observer comment l'altitude change.
4. Tester à l'extérieur : comparer avec l'altitude donnée par Google Maps.

DHT11: Simulation du Capteur

> *Tu vas apprendre à utiliser un **objet Python** qui simule un capteur de température et d'humidité (le DHT11). Cela te permettra d'écrire du code **exactement comme si tu avais un vrai capteur branché** sur un Raspberry Pi Pico.*

Utiliser la classe `DHT11` en Python

Tu dois d'abord télécharher le fichier Python ci-joint. Ensuite, crée un nouveau fichier python dans Thonny que tu appelles "simulation_dht11.py" et sauvegarde-le dans ton dossier "Téléchargements".

Étape 1 : importer le module

Quand on veut utiliser une classe qui se trouve dans un autre fichier Python, on doit **l'importer**.

👉 Imaginons que ton fichier avec la classe `DHT11` s'appelle `dht11.py`. Alors, tu dois écrire en haut de ton programme :

```
from dht11 import DHT11
```

Étape 2 : créer un objet capteur

Maintenant, tu peux **créer un capteur simulé**. Il faut lui donner le numéro de la broche (pin) sur laquelle il est censé être branché :

```
capteur = DHT11(pin=15)
```

Même si la simulation n'utilise pas vraiment la broche, on garde ce paramètre pour ressembler au vrai capteur.

Étape 3 : demander une mesure

Avant de lire la température ou l'humidité, tu dois **lancer une mesure** :

```
capteur.measure()
```

C'est comme si tu envoyais l'ordre au vrai capteur de "prendre une photo" de l'air ambiant.

Étape 4 : lire les valeurs

Une fois la mesure faite, tu peux demander :

```
print("Température :", capteur.temperature(), "°C")
print("Humidité :", capteur.humidity(), "%")
```

Chaque appel va te donner la **dernière valeur mesurée**.

Exemple complet

```
from dht11 import DHT11

# Création d'un capteur simulé
capteur = DHT11(pin=15)

# Faire 3 mesures
for i in range(3):
    capteur.measure()  # on lance une nouvelle mesure
    print("Température :", capteur.temperature(), "°C")
    print("Humidité :", capteur.humidity(), "%")
    print("---")
```

À retenir

1. On **importe** la classe depuis son fichier.
2. On **crée un objet** `DHT11`.
3. On appelle `measure()` pour lancer une mesure.
4. On récupère les valeurs avec `temperature()` et `humidity()`.

👉 Quand tu passeras au vrai Raspberry Pi Pico, tu n'auras presque rien à changer : seule la ligne `from dht11 import DHT11` sera remplacée par l'import du vrai module.

Veux-tu que je t'écrive aussi la **version avec le vrai module `dht` de MicroPython** pour montrer le passage simulation → réel ?

DHT11: Capteur de température et d'humidité

> *Cet article présente le DHT11, un capteur de température et d'humidité très répandu et facile à utiliser.*

- Le capteur DHT11 est un capteur d'humidité et de température numérique économique et couramment utilisé.
- Il fournit des lectures de température et d'humidité relativement précises à partir d'une seule broche de données.



Caractéristiques

Le capteur DHT11 est un capteur de température et d'humidité très couramment utilisé dans de nombreuses applications. Voici quelques-unes de ses caractéristiques principales :

1. **Mesure de la température et de l'humidité** : Le capteur DHT11 permet de mesurer à la fois la température et l'humidité relative de l'environnement dans lequel il est placé.
2. **Précision** : Le capteur DHT11 offre une **précision de ±2°C** pour la mesure de la température et de **±5%** pour la mesure de l'humidité relative.
3. **Plage de mesure** : La plage de mesure de la température du capteur DHT11 est généralement de **0°C à 50°C**, tandis que la plage de mesure de l'humidité relative est de **20% à 90%**.
4. **Alimentation** : Le capteur DHT11 fonctionne avec une tension d'alimentation de **3,3V à 5V**, ce qui le rend compatible avec de nombreux microcontrôleurs et cartes de développement.
5. **Sortie numérique** : Le capteur DHT11 utilise une **sortie numérique** pour transmettre les données de température et d'humidité. Il envoie les informations sous forme de signal série numérique.
6. **Temps de réponse** : Le capteur DHT11 a un **temps de réponse relativement lent** par rapport à d'autres capteurs de température et d'humidité. Il peut prendre plusieurs secondes pour effectuer une mesure et transmettre les données.
7. **Connectivité** : Le capteur DHT11 utilise un protocole de communication à **un seul fil**, ce qui le rend **facile à intégrer** dans des projets électroniques et des systèmes embarqués.

© 2025 Mr. Lambrechts

Il convient de noter que le capteur DHT11 est un **capteur d'entrée de gamme** largement utilisé pour des applications simples de mesure de température et d'humidité. Si une précision plus élevée ou une plus grande plage de mesure est requise, il existe d'autres capteurs plus avancés disponibles sur le marché (tels que le DHT22 ou le BME280)

DHT22: alternative plus précise

Parmi les modèles plus "haut de gamme", on citera par exemple le **DHT22**. Également connu sous le nom d'AM2302, il offre une **meilleure précision** et une **plus large plage de mesure** par rapport au DHT11. Il peut mesurer la température avec une précision de **±0,5°C** et l'humidité avec une précision de ±2%. De plus, le DHT22 a une plage de mesure de température de **-40°C à 80°C** et une plage d'humidité de **0% à 100%**.

Configuration matérielle

- Connectez la broche de données du capteur DHT11 au GPIO du Raspberry Pi Pico (par exemple, GPIO2).
- Alimentez le capteur DHT11 en connectant le fil rouge au 3,3 V du Pico et le fil noir à la masse (GND).

Installation de la bibliothèque DHT11

- Téléchargez la bibliothèque dht11 pour MicroPython à partir du dépôt GitHub officiel (https://github.com/mcauser/micropython-dht11).
- Copiez le fichier dht11.py sur votre Raspberry Pi Pico.

Exemple de code pour la lecture des données du capteur DHT11

```
import machine
import dht
import time

# Initialize the DHT sensor.
capteur = dht.DHT11(machine.Pin(2))

while True:
    capteur.measure()  # Trigger a measurement.
    print("Température:", capteur.temperature())
    print("Humidité:", capteur.humidity())
    time.sleep(2)
```

Explication du code

Bien sûr, voici une explication pour ce code :

- `import machine, import dht, import time` : Ces trois instructions importent les modules nécessaires. `machine` est un module qui permet d'interagir avec le matériel de l'appareil, `dht` est un module pour travailler avec le capteur de température et d'humidité DHT11, et `time` est un module pour manipuler le temps (par exemple, pour créer des délais).
- `capteur = dht.DHT11(machine.Pin(2))` : Cette ligne crée une instance de l'objet DHT11 appelée `capteur`. Le DHT11 est connecté au pin 2 de la machine (en supposant que vous utilisiez un ESP8266 ou un ESP32, où le pin 2 est généralement disponible).
- `while True:` : Ceci commence une boucle infinie. Le code à l'intérieur de cette boucle continuera à s'exécuter indéfiniment.
- `capteur.measure()` : Cette ligne indique au capteur DHT11 de commencer une mesure. Le capteur effectue une lecture de la température et de l'humidité de l'air.
- `print("Température:", capteur.temperature())` : Cette ligne affiche la température mesurée par le capteur. La méthode `temperature()` renvoie la température mesurée lors du dernier appel à `capteur.measure()`.
- `print("Humidité:", capteur.humidity())` : De même, cette ligne affiche l'humidité mesurée par le capteur. La méthode `humidity()` renvoie l'humidité mesurée lors du dernier appel à `capteur.measure()`.
- `time.sleep(2)` : Pour finir, cette ligne fait une pause dans l'exécution du programme pendant 2 secondes. C'est important car le capteur DHT11 a besoin d'un certain temps entre les lectures pour garantir des mesures précises. En général, il est recommandé d'attendre au moins 2 secondes entre chaque lecture.

Donc, en résumé, ce code effectue une lecture continue de la température et de l'humidité avec un capteur DHT11, et imprime ces valeurs toutes les 2 secondes. Ressources supplémentaires

Infos supplémentaires

- Tutoriel vidéo sur l'utilisation du capteur DHT11 avec MicroPython et Raspberry Pi Pico : https://youtu.be/hZ9zdvTRnTw
- Documentation de la bibliothèque DHT11 pour MicroPython : https://github.com/mcauser/micropython-dht11

Ces instructions vous permettront de lire les données de température et d'humidité à partir du capteur DHT11 à l'aide de MicroPython sur votre Raspberry Pi Pico.

HC-SR04: Capteur de distance à ultra-sons

> *Cet article présente HC-SR04, un capteur de distance à ultra-sons très répandu et facile à utiliser.*

Introduction au capteur de distance HC-SR04

- Le capteur de distance HC-SR04 est un capteur ultrasonique populaire utilisé pour mesurer la distance entre le capteur et un objet.
- Il utilise des ondes sonores ultrasoniques pour déterminer la distance, en émettant un signal et en mesurant le temps nécessaire pour que l'écho revienne.

Configuration matérielle

- Connectez la broche de déclenchement (Trig) du capteur HC-SR04 à un GPIO du Raspberry Pi Pico (par exemple, GPIO2).
- Connectez la broche d'écho (Echo) du capteur HC-SR04 à un autre GPIO du Pico (par exemple, GPIO3).
- Alimentez le capteur HC-SR04 en connectant le fil rouge au 5V du Pico et le fil noir à la masse (GND).

Exemple de code pour la mesure de distance avec le capteur HC-SR04

```
import machine
import utime

# Définir les broches GPIO pour le déclenchement et l'écho
trigger_pin = machine.Pin(2, machine.Pin.OUT)
echo_pin = machine.Pin(3, machine.Pin.IN)

# Fonction pour mesurer la distance
def mesure_distance():
    # Générer une impulsion de déclenchement
    trigger_pin.low()
    utime.sleep_us(2)
    trigger_pin.high()
    utime.sleep_us(10)
    trigger_pin.low()

    # Attendre le signal d'écho
    while echo_pin.value() == 0:
        signal_depart = utime.ticks_us()

    while echo_pin.value() == 1:
        signal_arrivee = utime.ticks_us()

    # Calculer la durée de l'écho
    duree_echo = signal_arrivee - signal_depart

    # Calculer la distance en fonction de la durée de l'écho
    distance = (duree_echo * 0.0343) / 2

    return distance

while True:
    # Mesurer la distance avec le capteur HC-SR04
    distance = mesure_distance()

    # Afficher la distance mesurée
    print("Distance : {:.2f} cm".format(distance))

    # Attendre avant de refaire une mesure
    utime.sleep(1)
```

Explication du code

- Importez les modules nécessaires, notamment machine pour accéder aux fonctionnalités du Raspberry Pi Pico et utime pour la gestion du temps.
- Définissez les broches GPIO pour le déclenchement (Trig) et l'écho (Echo) du capteur HC-SR04.
- Définissez une fonction mesure_distance() pour effectuer la mesure de distance.
- Dans la fonction mesure_distance(), générez une impulsion de déclenchement en mettant la broche de déclenchement à l'état bas puis haut puis à nouveau bas.
- Attendez le signal d'écho en mesurant le temps auquel l'écho commence et se termine.

Calculez la durée de l'écho en soustrayant le temps de départ du temps d'arrivée. 7. Calculez la distance en utilisant la formule de conversion de la durée de l'écho en distance (en supposant une vitesse du son de 343 m/s).

- Dans la boucle principale, appelez la fonction mesure_distance() pour mesurer la distance avec le capteur HC-SR04.
- Affichez la distance mesurée et attendez avant de refaire une mesure.

Utilisation du capteur de distance HC-SR04 avec un module open source

Installation du module open source

- Téléchargez le module open source hc_sr04 pour MicroPython à partir du dépôt GitHub officiel (https://github.com/rsc1975/micropython-hcsr04).
- Copiez les fichiers hc_sr04.py et pulseio.py sur votre Raspberry Pi Pico.

Exemple de code pour la mesure de distance avec le capteur HC-SR04

```
import machine
import utime
from hc_sr04 import HCSR04

# Définir les broches GPIO pour le déclenchement et l'écho
trigger_pin = machine.Pin(2)
echo_pin = machine.Pin(3)

# Créer une instance du capteur HC-SR04
capteur_distance = HCSR04(trigger_pin, echo_pin)

while True:
    # Mesurer la distance avec le capteur HC-SR04
    distance = capteur_distance.distance_cm()

    # Afficher la distance mesurée
    print("Distance : {:.2f} cm".format(distance))

    # Attendre avant de refaire une mesure
    utime.sleep(1)
```

Explication du code

- Importez les modules nécessaires, notamment machine pour accéder aux fonctionnalités du Raspberry Pi Pico et utime pour la gestion du temps.
- Importez la classe HCSR04 du module hc_sr04 pour utiliser le capteur HC-SR04.
- Définissez les broches GPIO pour le déclenchement (Trig) et l'écho (Echo) du capteur HC-SR04.
- Créez une instance du capteur HC-SR04 en passant les broches GPIO comme arguments.
- Dans la boucle principale, appelez la méthode distance_cm() de l'instance du capteur pour mesurer la distance.
- Affichez la distance mesurée et attendez avant de refaire une mesure.

Ressources supplémentaires

- Tutoriel vidéo sur l'utilisation du capteur de distance HC-SR04 avec MicroPython et Raspberry Pi Pico : https://youtu.be/YAF3o3SsCmk
- Dépôt GitHub du module open source hc_sr04 pour MicroPython : https://github.com/rsc1975/micropython-hcsr04

Ces instructions vous permettront de mesurer la distance à l'aide du capteur HC-SR04 en utilisant un module open source avec MicroPython sur votre Raspberry Pi Pico.

Ressources supplémentaires

- Tutoriel vidéo sur l'utilisation du capteur de distance HC-SR04 avec MicroPython et Raspberry Pi Pico : https://youtu.be/YAF3o3SsCmk

- Bibliothèque hcsr04 pour MicroPython sur GitHub : https://github.com/rsc1975/micropython-hcsr04

Ces instructions vous permettront de mesurer la distance à l'aide du capteur HC-SR04 en utilisant MicroPython sur votre Raspberry Pi Pico.

Les micro-contrôleurs

> *Un microcontrôleur est une unité de commande intégrée utilisée pour **contrôler le fonctionnement d'un robot**. Il s'agit d'un petit ordinateur sur une seule puce (SoC) qui combine un **microprocesseur**, de la **mémoire** et des **interfaces d'entrée/sortie**.*

En tant qu'unité de commande, le microcontrôleur **exécute les programmes et les instructions** qui définissent le comportement du robot. Il reçoit des informations provenant des capteurs du robot, traite ces données et envoie des commandes aux actionneurs pour effectuer des actions spécifiques.

Les microcontrôleurs sont souvent utilisés dans les robots en raison de leur **taille compacte**, de leur **faible consommation d'énergie** et de leur capacité à exécuter des tâches en **temps réel**. Ils offrent une solution intégrée pour contrôler les mouvements, les interactions avec l'environnement et les communications du robot.

Les microcontrôleurs peuvent être programmés à l'aide de différents langages, tels que **C/C++**, Python, **Micro-Python**, ou des langages de programmation visuels basés sur des blocs. Les programmeurs écrivent des codes qui spécifient le comportement du robot, y compris la prise de décision, les mouvements, la détection d'obstacles et les réponses aux événements.

En résumé, un microcontrôleur est l'**unité de commande centrale** d'un robot, responsable de l'exécution des programmes, de la gestion des capteurs et des actionneurs, et de la coordination des opérations pour permettre au robot d'accomplir des tâches spécifiques.

Micro:bit :

- Le micro:bit est un microcontrôleur largement utilisé pour l'apprentissage de la robotique, en particulier pour les débutants.
- Il est petit, abordable et **dispose de nombreux capteurs intégrés **tels que l'accéléromètre, le magnétomètre et les boutons.
- Le micro:bit utilise une programmation par blocs conviviale, ce qui le rend facile à apprendre pour les débutants.
- Il est souvent utilisé pour créer des projets robotiques simples tels que des robots suiveurs de ligne ou des robots éviteurs d'obstacles.

En savoir plus (notes de cours)

ESP8266 :

- L'ESP8266 est un microcontrôleur WiFi populaire utilisé dans la robotique et l'Internet des objets (IoT).
- Il dispose d'une connectivité WiFi intégrée, ce qui permet aux robots de se connecter à des réseaux sans fil et d'interagir avec d'autres appareils.
- L'ESP8266 est compatible avec plusieurs langages de programmation, tels que Arduino IDE et MicroPython.
- Il est souvent utilisé pour des projets de robotique avancés nécessitant une communication sans fil, tels que des robots contrôlés par smartphone ou des robots de surveillance.

Raspberry Pi Pico :

- Le Raspberry Pi Pico est un microcontrôleur développé par la Fondation Raspberry Pi.
- Il est basé sur le microcontrôleur RP2040 et dispose de nombreuses fonctionnalités, telles que des broches d'E/S polyvalentes et une programmation en MicroPython et C/C++.
- Le Raspberry Pi Pico est abordable, polyvalent et peut être utilisé pour des projets robotiques de taille moyenne à avancée.
- Il est souvent utilisé pour créer des robots autonomes, des robots avec des capteurs avancés et des projets de robotique embarquée.

En savoir plus (notes de cours)

LEGO Mindstorms :

- LEGO Mindstorms est une **plateforme de robotique éducative** très populaire pour les enfants et les débutants.
- Elle utilise des briques LEGO spéciales, des moteurs et des capteurs pour construire des robots.
- Les robots LEGO Mindstorms peuvent être programmés à l'aide d'un logiciel convivial basé sur des blocs graphiques.
- Cette plateforme est idéale pour enseigner les bases de la robotique et de la programmation, en mettant l'accent sur la créativité et la construction physique.

Thymio :

- Thymio est un **robot éducatif** conçu pour l'apprentissage de la robotique et de la programmation.
- Il est doté de capteurs et d'actionneurs, tels que des capteurs de proximité, des détecteurs de lumière et des moteurs.
- Thymio peut être programmé à l'aide d'un logiciel visuel ou d'un langage de programmation textuel, ce qui permet d'explorer différents niveaux de complexité.
- Ce robot est largement utilisé dans les écoles pour enseigner la programmation, la logique et les concepts de base de la robotique.

Ces microcontrôleurs offrent une excellente opportunité d'apprentissage pour les étudiants dans le domaine de la robotique. Ils sont abordables, faciles à programmer et offrent des fonctionnalités adaptées à différents niveaux de compétence.

Le Raspberry Pi Pico

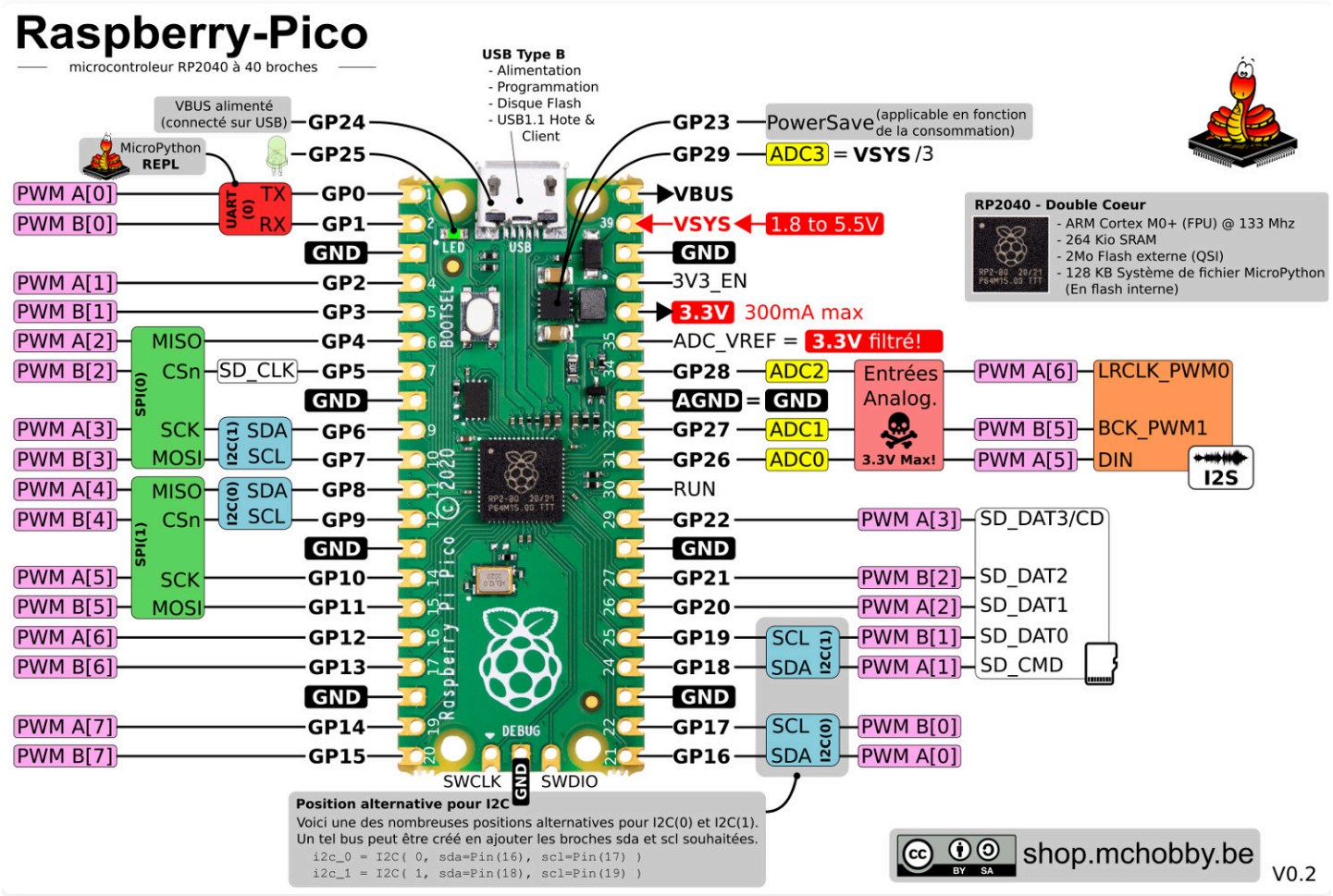
Les Broches du Raspberry Pi Pico en MicroPython

> *Sur cette page, nous allons explorer les différentes broches du Pico, leur numéro, leur fonction (analogique, PWM, numérique), et comment les utiliser en MicroPython.*

Le Raspberry Pi Pico dispose de plusieurs broches GPIO (General-Purpose Input/Output) qui peuvent être utilisées pour diverses tâches en MicroPython.

Broches GPIO du Raspberry Pi Pico

Le Raspberry Pi Pico possède 26 broches GPIO que vous pouvez utiliser pour interagir avec des composants électroniques et des capteurs. Voici une liste des broches GPIO disponibles sur le Pico :



Utilisation des Broches en MicroPython

Configuration d'une broche en mode numérique

Pour configurer une broche en mode numérique (entrée ou sortie), vous pouvez utiliser la bibliothèque `machine` en MicroPython. Par exemple, pour configurer la broche GP0 en tant que sortie, vous pouvez utiliser le code suivant :

```
import machine

# Configurer GP0 en sortie
broche_gp0 = machine.Pin(0, machine.Pin.OUT)
```

MicroPython offre plusieurs méthodes pour contrôler la tension de sortie :

```
from machine import Pin

Pin.on() et Pin.off()
Pin.high() et Pin.low()
Pin.value(valeur)
```

Les noms de fonctions sont explicites :

```
pin_led = Pin(16, mode=Pin.OUT)

# Impose une tension de 3.3V en sortie (état logique haut)
pin_led.on()
pin_led.high()
pin_led.value(1)

# Impose une tension de 0V en sortie (état logique bas)
pin_led.off()
pin_led.low()
pin_led.value(0)
```

Utilisation des broches en mode PWM

Le Raspberry Pi Pico prend en charge la modulation de largeur d'impulsion (PWM) sur certaines broches. Cela permet de contrôler la vitesse des moteurs, la luminosité des LED, etc. Voici comment configurer une broche en mode PWM :

```
import machine

# Configurer GP2 en mode PWM
broche_gp2 = machine.Pin(2)
pwm = machine.PWM(broche_gp2)
```

Lecture de valeurs analogiques

Pour lire des valeurs analogiques à partir des broches GPIO, vous pouvez utiliser l'ADC (Convertisseur Analogique-Numérique) intégré au Pico.

Note/Rappel: Le RPi Pico possède 4 broches analogiques: GP26->29

Voici comment configurer une broche pour lire une valeur analogique :

```
import machine

# Configurer GP26 pour la lecture analogique
broche_gp26 = machine.Pin(26)
adc = machine.ADC(broche_gp26)

# Lire la valeur analogique
valeur_analogique = adc.read_u16()
```

Le code suivant permet de lire la valeur d'un potentiomètre analogique:

```
from machine import ADC, Pin
from utime import sleep
pot = ADC(26)
conversion_factor = 3.3 / 65535

while True:
    raw = pot.read_u16()
    volts = raw * conversion_factor
    print('Raw: {} '.format(raw), 'Voltage {:.1f}V'.format(volts))
    sleep(1)
```

Et ce code permet de changer la fréquence de clignotement d'une LED en fonction de la valeur lue sur un potentiomètre:

```
from machine import ADC, Pin
from utime import sleep

pot = ADC(26)
led = Pin(25, Pin.OUT)

def map(s, a1, a2, b1, b2):
    return b1 + (s - a1) * (b2 - b1) / (a2 - a1)

while True:
    raw = pot.read_u16()
    delay = map(raw, 0, 65535, 0, 1)
    print('Raw: {} '.format(raw), 'Delay {:.1f}s'.format(delay))
    led.value(1)
    sleep(delay)
    led.value(0)
    sleep(delay)
```

Conclusion

Les broches GPIO du Raspberry Pi Pico offrent de nombreuses possibilités pour interagir avec le monde physique en utilisant MicroPython. Que ce soit pour contrôler des LED, des moteurs, ou pour lire des capteurs analogiques, vous avez maintenant les connaissances de base pour commencer à créer vos propres projets avec le Pico. Explorez davantage et expérimentez avec les différentes broches pour donner vie à vos idées.

MicroPython sur RPi Pico avec Thonny Python

> ///

Prérequis

- Assurez-vous d'avoir un Raspberry Pi Pico fonctionnel et un câble USB pour le connecter à votre ordinateur.
- Téléchargez et installez l'IDE Thonny Python sur votre ordinateur à partir du site officiel de Thonny (https://thonny.org/).

Installation du firmware MicroPython

- Visitez le site officiel de MicroPython pour Raspberry Pi Pico (https://micropython.org/download/rp2-pico/) et téléchargez le dernier firmware MicroPython (.uf2) pour le Raspberry Pi Pico.

Connexion du Raspberry Pi Pico

- Connectez votre Raspberry Pi Pico à votre ordinateur à l'aide du câble USB. Assurez-vous que le commutateur "BOOTSEL" est enfoncé pour mettre le Pico en mode Bootloader.

Flashage du firmware MicroPython

- Ouvrez l'IDE Thonny Python sur votre ordinateur.
- Dans Thonny, cliquez sur le bouton "Sélectionner l'interpréteur" en haut à droite.
- Dans la fenêtre de sélection de l'interpréteur, choisissez "Raspberry Pi Pico (MicroPython)".
- Maintenant, dans Thonny, cliquez sur le bouton "Flasher" en haut.
- Dans la fenêtre de flashage, sélectionnez le firmware MicroPython (.uf2) que vous avez précédemment téléchargé.
- Cliquez sur "OK" pour flasher le firmware sur votre Raspberry Pi Pico.

Écriture et exécution du code MicroPython

- Une fois le firmware MicroPython installé, le Raspberry Pi Pico redémarrera automatiquement.
- Dans Thonny, vous verrez le Pico répertorié comme un périphérique connecté.
- Créez un nouveau fichier en cliquant sur "Nouveau" dans le menu Fichier.
- Écrivez votre code MicroPython dans le nouvel onglet du fichier.
- Pour exécuter le code, cliquez sur le bouton "Exécuter" ou appuyez sur F5.

Interaction avec le Raspberry Pi Pico

- Vous pouvez utiliser la console REPL (Read-Eval-Print Loop) pour interagir avec le Raspberry Pi Pico en temps réel.
- Dans Thonny, cliquez sur l'icône de console en bas de la fenêtre pour ouvrir la console REPL.
- Vous pouvez saisir des commandes directement dans la console pour interagir avec les broches GPIO, les capteurs ou les actionneurs connectés au Raspberry Pi Pico.

Débogage et surveillance

- Thonny Python offre également des fonctionnalités de débogage et de surveillance pour vous aider à identifier et corriger les erreurs dans votre code MicroPython.
- Vous pouvez placer des points d'arrêt, examiner les variables et suivre l'exécution pas à pas pour comprendre le comportement de votre code.

Ces étapes vous permettront de configurer et d'utiliser MicroPython sur votre Raspberry Pi Pico en utilisant l

Raspberry Pi Pico et les Ports GPIO

///

- Le Raspberry Pi Pico est un microcontrôleur à faible coût développé par la Fondation Raspberry Pi.
- Il est basé sur le microcontrôleur RP2040 et offre de nombreuses fonctionnalités pour les projets de robotique et d'électronique.



Les Ports GPIO (General-Purpose Input/Output)

- Le Raspberry Pi Pico dispose de **26 broches GPIO (0 à 25)** qui peuvent être configurées en tant qu'**entrées ou sorties** pour interagir avec d'autres composants électroniques.
- Les broches GPIO peuvent être utilisées pour **lire des signaux provenant de capteurs externes ou pour contrôler des actionneurs** tels que des moteurs ou des LED.
- Chaque broche GPIO peut être configurée individuellement et prend en charge différents modes de fonctionnement, tels que l'**entrée**, la **sortie**, la **PWM** (modulation de largeur d'impulsion), l'I2C, le SPI, etc.

Utilisation des Ports GPIO

- Configuration des broches : Avant d'utiliser une broche GPIO, il est nécessaire de la **configurer dans le mode de fonctionnement souhaité** en utilisant des bibliothèques de programmation appropriées, telles que RP2PIO ou RP2GPIO pour le Raspberry Pi Pico.
- Lecture des broches d'entrée : Les broches GPIO configurées en tant qu'entrées peuvent être utilisées pour lire des signaux provenant de capteurs externes. On peut les lire en utilisant des fonctions de lecture appropriées dans le langage de programmation utilisé, telles que input() en Python.
- Contrôle des broches de sortie : Les broches GPIO configurées en tant que sorties peuvent être utilisées pour contrôler des actionneurs, tels que des moteurs ou des LED. On peut les contrôler en utilisant des fonctions d'écriture appropriées dans le langage de programmation utilisé, telles que output() en Python.

Exemples d'utilisation des Ports GPIO

- Contrôle de LED : On peut connecter une LED à une broche GPIO configurée en tant que sortie et utiliser la fonction d'écriture pour allumer ou éteindre la LED.
- Lecture de bouton : On peut connecter un bouton à une broche GPIO configurée en tant qu'entrée et utiliser la fonction de lecture pour détecter si le bouton est enfoncé ou relâché.
- Contrôle de moteur : On peut utiliser les broches GPIO configurées en tant que sorties pour contrôler la vitesse et la direction d'un moteur en utilisant la modulation de largeur d'impulsion (PWM) pour créer des signaux analogiques simulés.

Modes Analogique, Digital et PWM

Mode Analogique

- Le mode analogique est utilisé pour représenter et manipuler des **signaux continus** qui peuvent prendre une gamme de valeurs différentes.
- Dans le contexte des microcontrôleurs, le mode analogique permet de **mesurer ou de générer des signaux analogiques**.
- Lorsqu'un microcontrôleur est configuré en mode analogique, il peut **lire des tensions** provenant de capteurs analogiques tels que des capteurs de température ou de lumière.
- Les microcontrôleurs peuvent également générer des signaux analogiques en utilisant des *convertisseurs numérique-analogique* (CNA) pour contrôler des dispositifs tels que des amplificateurs ou des moteurs.

Mode Digital

- Le mode digital est utilisé pour représenter des signaux qui ne peuvent prendre que **deux valeurs distinctes : 0 et 1 (ou HIGH et LOW)**.
- Les microcontrôleurs sont capables de traiter des signaux digitaux, ce qui leur permet de communiquer avec d'autres composants électroniques tels que des capteurs, des actionneurs ou d'autres microcontrôleurs.
- Les broches GPIO configurées en tant que sorties digitales peuvent envoyer des signaux à d'autres composants, tandis que les broches configurées en tant qu'entrées digitales peuvent recevoir des signaux provenant d'autres sources.

Modulation de Largeur d'Impulsion (PWM)

- La modulation de largeur d'impulsion (PWM) est une technique utilisée pour **simuler des signaux analogiques en utilisant des signaux digitaux**.
- Elle est couramment utilisée pour contrôler la vitesse ou l'intensité de dispositifs tels que des moteurs, des LED ou des servomoteurs.
- Dans un signal PWM, **la largeur de l'impulsion varie dans le temps**, tandis que la période de l'impulsion reste constante.

- En ajustant le rapport cyclique (rapport entre la durée de l'impulsion et la période), on peut contrôler la moyenne pondérée du signal, ce qui permet de varier la vitesse ou l'intensité du dispositif contrôlé.

Utilisation dans la Robotique

- En robotique, les **signaux analogiques** sont souvent utilisés **pour mesurer des grandeurs** telles que la **température**, la **luminosité** ou la **distance**.
- Les signaux digitaux sont utilisés pour la communication entre composants électroniques et la **lecture d'états** de capteurs, tels que les **boutons** ou les **interrupteurs**.
- La modulation de largeur d'impulsion (**PWM**) est utilisée pour **contrôler précisément la vitesse des moteurs** ou la **luminosité des LED**, ce qui est essentiel dans de nombreux projets robotiques.

Il est important de comprendre ces différents modes et techniques, car ils permettent aux microcontrôleurs de s'adapter à différentes situations et de contrôler divers dispositifs dans les projets robotiques.

Ressources supplémentaires

- Documentation officielle du Raspberry Pi Pico : https://www.raspberrypi.org/documentation/pico/
- Tutoriels et exemples de projets : Le site officiel du Raspberry Pi Pico propose une variété de tutoriels et de projets pour vous aider à démarrer avec les ports GPIO et d'autres fonctionnalités du microcontrôleur.

Utilisation du Raspberry Pi Pico avec MicroPython

> *Dans cet article de cours, nous allons explorer comment utiliser le Raspberry Pi Pico avec MicroPython et vous montrer des exemples de programmes pour lire et écrire des valeurs sur les broches GPIO (General-Purpose Input/Output).*

Le Raspberry Pi Pico est un microcontrôleur puissant et abordable qui peut être programmé avec MicroPython, un langage de programmation convivial pour les débutants.

Prérequis

Avant de commencer, assurez-vous d'avoir les éléments suivants à votre disposition :

- Un Raspberry Pi Pico
- Un ordinateur avec MicroPython installé (vous pouvez télécharger MicroPython sur le site officiel)
- Un câble micro USB pour la connexion entre l'ordinateur et le Raspberry Pi Pico

Installation de MicroPython

Ceci est une procédure d'installation manuelle que nous n'avons pas vue en classe. Je la laisse pour que vous sachiez que ça existe, mais il ne faut pas la connaître pour le cours.

1. Téléchargez la dernière version de MicroPython depuis le site officiel (https://micropython.org/).
2. Connectez votre Raspberry Pi Pico à votre ordinateur à l'aide du câble micro USB. Le Pico devrait être reconnu comme un périphérique de stockage USB.
3. Copiez le fichier MicroPython que vous avez téléchargé sur le Pico. Il doit avoir une extension ".uf2".
4. Déconnectez le Pico de votre ordinateur, puis reconnectez-le. Vous verrez un nouveau périphérique de stockage appelé "RPI-RP2".
5. Supprimez le fichier "CURRENT.UF2" s'il existe et renommez votre fichier MicroPython téléchargé en "CURRENT.UF2". Cela permettra au Pico de démarrer automatiquement avec MicroPython.
6. Déconnectez à nouveau le Pico de l'ordinateur, puis reconnectez-le. Il devrait maintenant exécuter MicroPython.

Programme de base

Commençons par un exemple simple de programme en MicroPython pour allumer et éteindre une LED connectée à une broche GPIO du Raspberry Pi Pico.

```
import machine
import utime

# Définir la broche GPIO à utiliser
led = machine.Pin(25, machine.Pin.OUT)

# Boucle infinie pour clignoter la LED
while True:
    led.toggle()  # Inverse l'état de la LED
    utime.sleep(1)  # Attendre 1 seconde
```

Dans ce programme, nous utilisons le module `machine` pour gérer les broches GPIO. Nous définissons la broche 25 comme une sortie (`machine.Pin.OUT`). Ensuite, nous utilisons une boucle infinie pour alterner l'état de la LED entre allumé et éteint toutes les secondes à l'aide de la méthode `toggle()` .

Lecture de valeurs analogiques

Vous pouvez également lire des valeurs analogiques à partir de broches GPIO pour mesurer des grandeurs telles que la luminosité avec un capteur de lumière. Voici un exemple pour lire une valeur analogique à partir de la broche 26 :

```
import machine

# Définir la broche GPIO à utiliser
analog_pin = machine.Pin(26)

# Configurer l'ADC (Convertisseur Analogique-Numérique)
adc = machine.ADC(analog_pin)

# Lire et imprimer la valeur analogique
while True:
    valeur = adc.read_u16()  # Lecture de la valeur analogique (0-65535)
    print("Valeur analogique : ", valeur)
```

Dans cet exemple, nous utilisons le module `machine` pour configurer l'ADC (Convertisseur Analogique-Numérique) sur la broche 26. La valeur analogique est lue à l'aide de la méthode `read_u16()` et affichée dans la console.

Conclusion

Le Raspberry Pi Pico avec MicroPython offre une excellente plateforme pour l'apprentissage de la programmation et de l'électronique. Vous pouvez créer une variété de projets intéressants en utilisant les broches GPIO pour contrôler des composants électroniques. N'hésitez pas à explorer davantage et à créer vos propres projets en utilisant ces concepts de base.

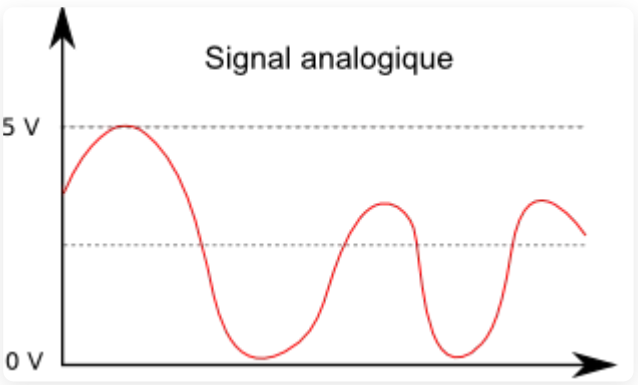
Les différents types de signaux

Les microcontrôleurs sont des composants essentiels dans de nombreux systèmes électroniques, des appareils ménagers aux robots industriels. Pour interagir avec leur environnement, ces microcontrôleurs utilisent différents types de signaux. Dans cet article, nous allons explorer en détail les trois principaux types de signaux utilisés : les signaux analogiques, les signaux digitaux et les signaux à modulation de largeur d'impulsion (PWM). Nous verrons comment ils fonctionnent, leurs applications et les bonnes pratiques pour les utiliser.

1. Signaux Analogiques

Qu'est-ce qu'un Signal Analogique ?

Un signal analogique est un **signal continu** qui peut prendre une **infinité de valeurs** dans une plage donnée. Contrairement aux signaux digitaux, qui ne peuvent prendre que des valeurs discrètes (0 ou 1), les signaux analogiques peuvent représenter des variations continues, comme la température, la luminosité ou le son.



Exemple de Signal Analogique

Imaginez un capteur de température analogique. La sortie de ce capteur est une tension continue qui varie en fonction de la température mesurée. Par exemple, à 20°C, la sortie pourrait être de 2V, à 25°C, elle pourrait être de 2.5V, et ainsi de suite.

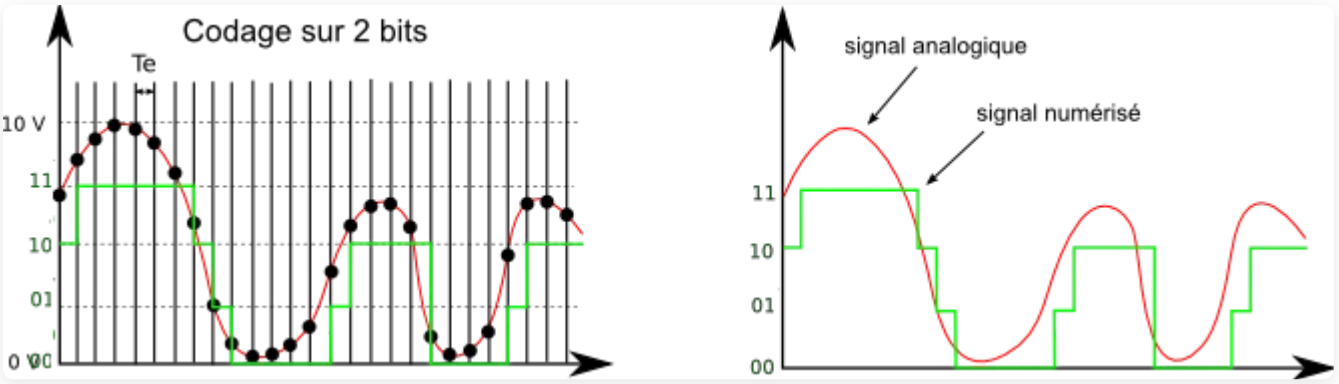
Utilisation dans les Microcontrôleurs

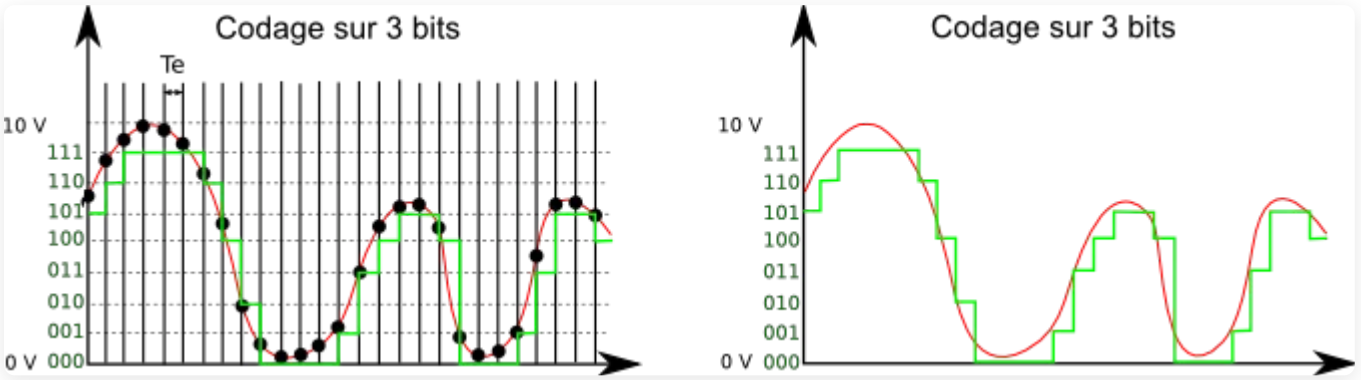
Les microcontrôleurs ne peuvent pas traiter directement les signaux analogiques. Ils doivent **convertir ces signaux** en valeurs numériques à l'aide d'un **convertisseur analogique-numérique (ADC)**. Une fois le signal converti, le microcontrôleur peut le traiter de manière numérique.

L'ADC **convertit une tension analogique continue** en une valeur numérique discrète que le microcontrôleur peut traiter. Cette conversion se fait en **échantillonnant** le signal analogique à des intervalles de temps réguliers et en attribuant à chaque échantillon une valeur numérique correspondante.

Chaque échantillon est attribué à la valeur numérique la plus proche parmi un ensemble de valeurs discrètes possibles.

Une caractéristique important des ADC est leur **résolution** : C'est le **nombre de bits de la sortie numérique**. Une résolution de **10 bits** signifie que la sortie peut prendre **1024** valeurs différentes (2^10). **Plus la résolution est élevée, plus la conversion est précise**.

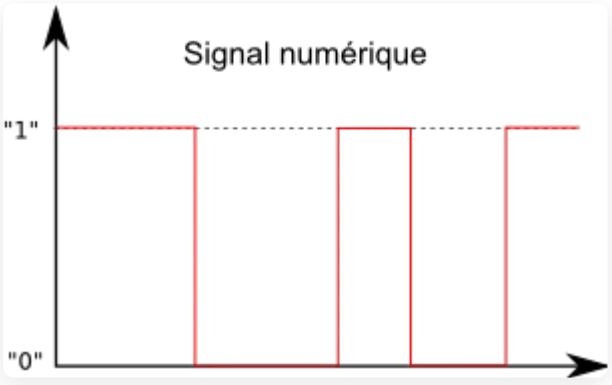




2. Signaux Digitaux

Qu'est-ce qu'un Signal Digital ?

Un signal **digital**, ou **numérique**, ne peut prendre que deux valeurs distinctes : **0 (faible) ou 1 (élevé)** (en tout cas en ce qui concerne les i/o des micro-controlleurs). Ces signaux sont utilisés pour représenter des états binaires ou des données discrètes.



Exemple de Signal Digital

Considérons un **interrupteur** connecté à un microcontrôleur. Lorsque l'interrupteur est fermé, la sortie est à 1 (élevé), et lorsqu'il est ouvert, la sortie est à 0 (faible).

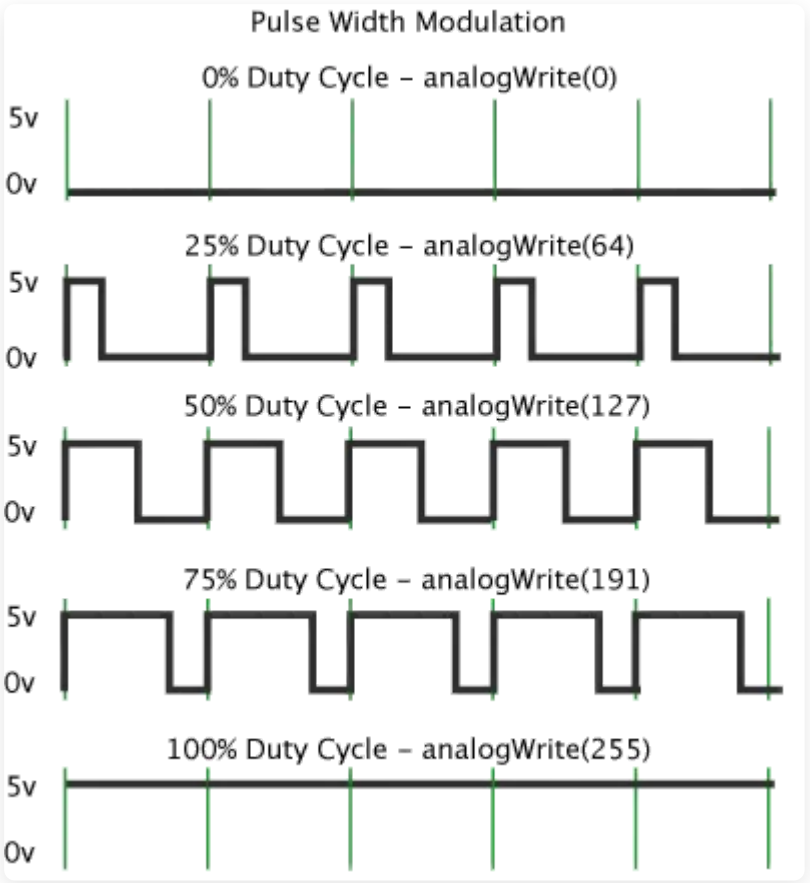
Utilisation dans les Microcontrôleurs

Les microcontrôleurs traitent naturellement les signaux digitaux. Ils peuvent lire les états des broches d'entrée/sortie (I/O) et exécuter des actions en fonction de ces états.

3. Signaux à Modulation de Largeur d'Impulsion (PWM)

Qu'est-ce que le PWM ?

La **modulation de largeur d'impulsion** (GB ***PWM ou pulse width modulation***) est une technique où la largeur des impulsions d'un signal est modulée pour représenter une **valeur analogique moyenne**. Le **rapport cyclique** (GB ***duty cycle***) détermine la **proportion de temps** pendant laquelle le signal est à l'état haut (1) par rapport à l'état bas (0).



Le graphe ci-dessus montre l'équivalence entre différents duty cycles et la valeur équivalente en analogique (analogWrite).

Exemple de PWM

Supposons que nous ayons une LED contrôlée par un signal PWM. En modifiant le rapport cyclique, nous pouvons contrôler la luminosité de la LED. Un rapport cyclique de 50% signifie que le signal est haut pendant 50% du temps et bas pendant les 50% restants, ce qui correspond à une luminosité moyenne de la LED.

Utilisation dans les Microcontrôleurs

Les microcontrôleurs peuvent générer des signaux PWM à l'aide de modules dédiés. Ces signaux sont souvent utilisés pour le contrôle de la puissance, comme dans les moteurs, les LEDs ou les régulateurs de tension.

Conclusion

Les microcontrôleurs utilisent des signaux **analogiques**, **digitaux** et **PWM** pour interagir avec leur environnement. Chaque type de signal a ses propres caractéristiques, avantages et inconvénients. Les signaux analogiques permettent de représenter des valeurs continues, les signaux digitaux sont parfaits pour les états binaires et la communication, et les signaux PWM offrent une solution efficace pour le contrôle de puissance et la génération de valeurs analogiques moyennes.

En comprenant comment utiliser ces différents types de signaux, vous pouvez concevoir des systèmes plus robustes et efficaces.



INTELLIGENCE ARTIFICIELLE

Chatbot, assistant intégré, agent : qui fait quoi ?

> *ChatGPT, Claude Code, GitHub Copilot, Cursor, Gemini, Operator… Tous reposent sur des modèles de langage, mais ils ne se ressemblent pas du tout à l'usage. Ce cours pose les trois grandes familles d'outils IA, ce que chacune sait faire, et comment choisir selon ton besoin — pour le code et au-delà.*

Quand on parle d'« IA », on parle souvent de **ChatGPT**. Mais ChatGPT n'est qu'une **catégorie** d'outil IA parmi d'autres. La même technologie sous-jacente (un modèle de langage, ou *LLM*) peut être emballée de **plusieurs façons** très différentes, qui ne servent pas du tout aux mêmes besoins.

Comprendre ces différences, c'est éviter trois pièges courants :

- utiliser un **chatbot** pour une tâche qui demanderait un **agent** → tu passes la moitié du temps en copier-coller
- utiliser un **agent** pour une tâche que résoudrait un **chatbot** → tu déploies un bulldozer pour planter un clou
- choisir un outil sur sa **hype** sans regarder ce qu'il fait vraiment → tu paies pour des fonctionnalités que tu n'utilises pas

---

🧠 Le point commun : un modèle de langage

Sous le capot, tous ces outils s'appuient sur un **LLM** (*Large Language Model*) — un modèle entraîné à prédire le texte le plus probable à partir d'un contexte. GPT-4, Claude Opus, Gemini, Llama, Mistral… ce sont des LLM.

Un LLM brut, c'est juste une fonction : on lui donne du texte, il complète. Pour devenir un outil utilisable, il faut l'**emballer** dans une interface. C'est cet **emballage** qui définit la catégorie d'outil — pas le modèle.

> 💡 *Le même modèle (par exemple Claude Sonnet 4.6) peut alimenter **un chatbot** (Claude.ai), **un assistant intégré** (Cursor) ou **un agent** (Claude Code). C'est le **type d'interface** qui change, pas l'intelligence du modèle.*

🗨️ Famille 1 — Le chatbot

Ce que c'est

Une **interface de conversation** dans un navigateur ou une appli mobile. Tu écris, le modèle répond. Rien ne sort de cette fenêtre — pour appliquer un résultat, **c'est à toi de copier-coller**.

EXEMPLES	URL
ChatGPT	chatgpt.com
Claude	claude.ai
Gemini	gemini.google.com
Le Chat (Mistral)	chat.mistral.ai
Perplexity	perplexity.ai

Ce qu'il fait bien

- **Expliquer** un concept (en cours, à la maison, sur un sujet inconnu)
- **Brainstormer** une idée, une structure, un plan
- **Rédiger** un texte, un email, une dissertation, un script
- **Traduire** ou **reformuler**
- **Analyser** un document ou une image que tu lui colles
- **Réviser** : tu lui demandes de te poser des questions sur un cours
- **Coder** par bouts copiés-collés (snippet par snippet)

Ses limites

- N'a **aucun accès** à ton disque, ton terminal, internet (sauf option *web search*), tes outils
- Tout résultat doit être **copié-collé manuellement** vers ta vraie destination
- La conversation se perd quand tu fermes l'onglet (sauf si tu sauvegardes)
- Tu dois lui **donner** le contexte à chaque fois (ses derniers échanges sont sa seule mémoire)

Quand l'utiliser

> *Quand le travail final est **du texte que tu vas relire et utiliser à la main**.*

Exemple : tu prépares un exposé, tu veux une explication claire d'un concept, ou tu demandes une review de ta dissertation. Le chatbot est parfait.

Famille 2 — L'assistant intégré (autocomplétion / inline)

Ce que c'est

L'IA est **intégrée dans un outil que tu utilises déjà** (ton éditeur de code, ton traitement de texte, ton mail). Elle te propose des **complétions** au fil de la frappe, ou répond à des questions **sans quitter** l'application.

EXEMPLES	INTÉGRÉ DANS
GitHub Copilot	VS Code, JetBrains, Visual Studio
Cursor (chat + tab)	Son propre éditeur (fork de VS Code)
Windsurf	Son propre éditeur
JetBrains AI	IntelliJ, Rider, PyCharm…
Microsoft 365 Copilot	Word, Excel, Outlook, Teams
Notion AI	Notion
Gmail Smart Compose	Gmail

Ce qu'il fait bien

- **Suggérer la suite** de ce que tu écris (variable, fonction, paragraphe…)
- **Compléter du code** répétitif : boucles, getters/setters, tests évidents
- **Répondre à une question** sur ton fichier ouvert sans changer d'app
- **Rester rapide** : pas besoin de prompt long, tu travailles en flot

Ses limites

- Vision **locale**, souvent limitée au fichier ouvert ou aux fichiers proches
- Ne **prend pas l'initiative** : c'est toi qui pilote ligne par ligne
- Peut **inventer** une API inexistante si le contexte est trop court (*hallucination*)
- Lié à un outil précis — si tu changes d'éditeur, tu perds l'assistant

Quand l'utiliser

> *Quand tu veux **garder le contrôle ligne par ligne** mais accélérer la frappe et éviter les répétitions.*

Exemple : tu codes une fonction, Copilot devine la suite et te la propose en gris — `Tab` pour accepter. Ou : tu écris un mail dans Gmail, l'IA complète ta phrase.

🤖 Famille 3 — L'agent

Ce que c'est

L'IA **agit** dans ton environnement. Elle lit des fichiers, exécute des commandes, modifie du code, fait des appels réseau — sous ton contrôle, mais **de sa propre initiative** quand tu lui donnes un objectif de haut niveau.

EXEMPLES (CODE)	PLATEFORME
Claude Code	Terminal, Anthropic
Codex CLI	Terminal, OpenAI
Gemini CLI	Terminal, Google
Cline / **Roo Code**	VS Code (open source)
Aider	Terminal (open source)

EXEMPLES (HORS CODE)	ACTION
Claude Computer Use	Contrôle ton bureau (souris, clavier, screenshots)
OpenAI Operator	Pilote un navigateur web pour faire des tâches
Browser Use	Bibliothèque open source, agents web
n8n / Make + IA	Automatisations complexes avec étapes IA

Ce qu'il fait bien

- **Tâches multi-étapes** où l'ordre dépend des résultats intermédiaires
- **Modifier plusieurs fichiers** de façon cohérente dans un projet
- **Lancer des commandes** (tests, build, git, scripts shell)
- **Naviguer sur le web** et accomplir des tâches concrètes (remplir un formulaire, comparer des prix…)
- **Itérer seul** : essayer, lire l'erreur, corriger, retenter

Ses limites

- Peut faire des **bêtises irréversibles** s'il n'est pas surveillé (`rm` , push forcé, suppression de données…)
- **Plus lent** par opération qu'un chatbot ou Copilot (beaucoup d'appels au modèle pour une tâche)
- **Consomme beaucoup** de quota / d'API (un agent peut faire 50 appels là où un chatbot en fait 1)
- Demande une **vraie supervision** : tu dois relire ce qu'il fait, pas juste accepter

Quand l'utiliser

> *Quand tu peux **décrire un objectif** plutôt qu'une séquence d'étapes, et que tu acceptes de **superviser** ce qui se passe.*

Exemple : *« ajoute un bouton Copier au générateur de mot de passe, écris un test, lance les tests »*. L'agent ouvre les fichiers, modifie, lance les tests, lit la sortie, ajuste. Tu valides à la fin.

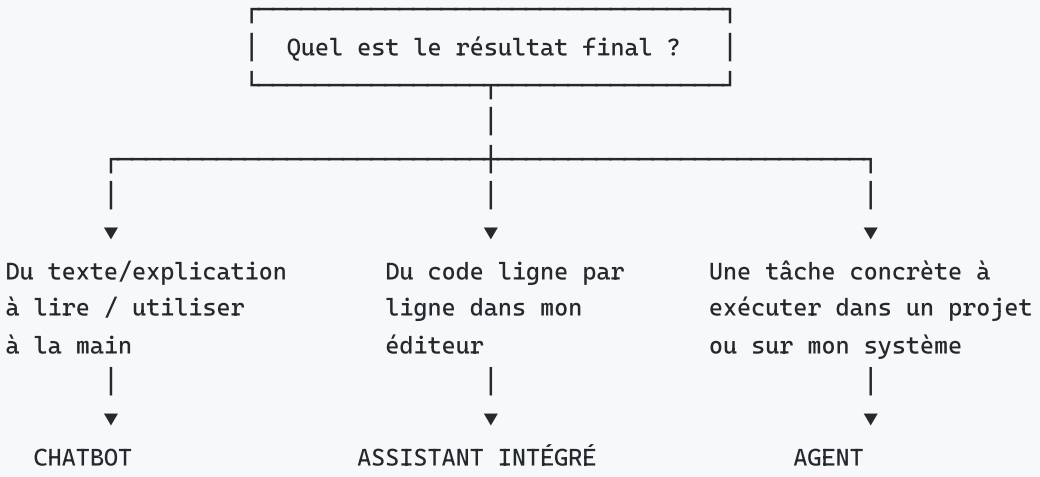
📊 Tableau comparatif

CRITÈRE	CHATBOT	ASSISTANT INTÉGRÉ	AGENT
Interface	Navigateur / appli	Dans ton outil	Terminal / IDE
Touche à ton système ?	Non	Édition locale ciblée	Oui, fichiers + commandes
Prise d'initiative	Aucune	Faible	Forte
Granularité	Échange par échange	Ligne par ligne	Tâche complète
Vitesse perçue	Rapide	Très rapide (inline)	Plus lent (multi-étapes)
Coût (intensif)	Faible	Moyen	Élevé
Risque	Quasi nul	Faible	Réel (à superviser)
Mémoire du projet	Toi tu la fournis	Fichier ouvert + proches	Lit le codebase

---

🎯 Comment choisir : la question à se poser

Avant de sortir un outil, demande-toi :



▸ Quelques cas concrets

BESOIN	BON OUTIL
Comprendre une notion de cours	**Chatbot** (ChatGPT, Claude)
Réviser pour un examen	**Chatbot** (génère un quiz, corrige-moi)
Rédiger un mail délicat	**Assistant intégré** (Gmail/Outlook) ou **chatbot**
Écrire une fonction `triBulle` en C#	**Assistant intégré** (Copilot, Rider AI)
Refactoriser un projet sur plusieurs fichiers	**Agent** (Claude Code, Codex CLI)
Faire passer des tests qui échouent	**Agent**
Comparer 3 produits sur des sites différents	**Agent web** (Operator, Browser Use)
Traduire un texte	**Chatbot**
Résumer un PDF de 50 pages	**Chatbot** (avec upload)
Générer 50 fichiers à partir d'un template	**Agent**
Compléter une formule Excel compliquée	**Assistant intégré** (Copilot Excel)
Brainstormer un sujet de TFE	**Chatbot**
Mettre à jour les imports cassés après un rename	**Agent**

🌍 Et au-delà du code ?

Les trois familles existent pour **tous les domaines**, pas juste le code :

- **Recherche / analyse** : un **chatbot** comme Perplexity cite ses sources ; un **agent** comme Operator peut naviguer pour toi.
- **Écriture** : Notion AI (intégré), ChatGPT (chatbot), agents de rédaction long-format (encore expérimentaux).
- **Bureautique** : Microsoft 365 Copilot, Google Duet AI — intégrés à Word, Excel, Sheets…
- **Image / vidéo** : DALL·E, Midjourney, Sora — ce sont des modèles spécialisés, mais l'**emballage** suit les mêmes catégories (interface chat ou intégrée dans un outil).
- **Vie quotidienne** : agents qui planifient des réservations, comparent des assurances, remplissent des formulaires administratifs.

Le réflexe est toujours le même : **quel est le résultat que je veux ? Faut-il que l'IA agisse, ou juste qu'elle me réponde ?**

💸 Et les coûts ?

FAMILLE	MODÈLE ÉCONOMIQUE TYPIQUE
Chatbot	Gratuit avec quotas, abonnement ~20 €/mois pour les versions Pro
Assistant intégré	Abonnement (Copilot : ~10 €/mois ; Cursor : ~20 €/mois ; gratuit pour les étudiants chez certains)
Agent	Soit abonnement (Claude Code Pro), soit **paiement à l'usage** via API (peut grimper vite — un projet long peut coûter quelques €)

> 💡 ***Astuce étudiant** : GitHub Copilot est **gratuit avec un compte étudiant** (GitHub Student Pack). Les chatbots ont tous une version gratuite suffisante pour l'usage scolaire. Pour les agents, les comptes gratuits ont des quotas qui suffisent à découvrir.*

⚠️ Ce que ces outils ne sont pas

Avant de tomber dans le piège du marketing :

- **Pas une source de vérité.** Tous peuvent **halluciner** — inventer une fonction, une date, un nom de loi. Vérifie ce qui compte.
- **Pas un substitut à la compréhension.** Si tu rends du code que tu ne comprends pas, tu n'apprends rien et tu seras coincé au premier bug.
- **Pas une boîte noire à laquelle déléguer.** Tu restes responsable de ce qui sort de chez toi.
- **Pas conscient.** Le modèle ne « comprend » pas comme un humain — il prédit du texte plausible. Ça suffit pour énormément de tâches, mais ça ne pense pas.

✍️ À retenir

- Tous ces outils reposent sur des **LLM**, mais leur **emballage** détermine ce qu'ils savent faire.
- **Chatbot** : on discute, on copie-colle. Idéal pour expliquer, rédiger, brainstormer.
- **Assistant intégré** : il complète ce que tu fais dans ton outil. Idéal pour rester rapide ligne par ligne.
- **Agent** : il agit dans ton projet ou ton système. Idéal pour les tâches multi-étapes décrites en langage naturel.
- **Bon outil = bon résultat avec le moins d'effort**. Réfléchis au type de résultat avant de choisir.
- Les trois familles existent **hors du code** : bureautique, recherche, image, vie quotidienne.
- Tu restes **responsable** de ce que tu produis avec ces outils — toujours.

Suite

L'article suivant (Coder son projet avec un agent IA) te fait passer à la pratique : installer un agent sur ta machine et l'utiliser pour de vrai sur un projet.

Exercice — Coder son projet avec un agent IA

> *Les agents IA de code (Claude Code, Codex, Gemini CLI…) ne sont plus de la science-fiction : ils écrivent, modifient et exécutent du code dans ton terminal, sous tes ordres. Ton job ici : en installer un sur ta machine, l'utiliser sur un vrai projet, et garder la maîtrise de ce qui est produit.*

🎯 Objectif

Tu vas installer **un agent IA de codage** sur ta machine, le connecter à un projet que tu écris (ou un projet existant), et t'en servir pour réaliser une fonctionnalité concrète. Tu rédigeras ensuite un court rapport sur ce que tu as appris, ce qui t'a impressionné, et **ce que l'agent a raté**.

> ⚠️ ***À l'école, seuls les comptes gratuits sont autorisés.*** *Pas d'abonnement payant, pas de carte de crédit. C'est volontaire — l'idée n'est pas de te faire dépenser, mais de te faire découvrir ce qui est accessible à tout le monde aujourd'hui.*

🤖 De quoi on parle exactement

Un **agent IA de code** est un programme qui tourne dans ton terminal (ou ton IDE) et qui :

- comprend ton **codebase** (lit les fichiers, suit les imports…)
- exécute des **commandes shell** (lance les tests, build, git…)
- modifie ou crée des **fichiers** selon tes instructions en langage naturel
- te demande confirmation pour les actions risquées

Ce n'est pas un chatbot qui te recrache du code à copier-coller. C'est un **outil qui agit** dans ton projet, sous ton contrôle.

▸ Les trois agents que tu peux installer

AGENT	ÉDITEUR	COMPTE GRATUIT
Claude Code	Anthropic	Oui, avec quotas
Codex CLI	OpenAI	Oui, avec quotas
Gemini CLI	Google	Oui, généreux quotas

À toi de choisir lequel installer. Les trois ont des qualités et des défauts. **Tu peux aussi en tester plusieurs**, c'est encouragé.

📔 Travail demandé

Partie 1 — Installation et configuration

1. **Choisis** un agent parmi Claude Code, Codex CLI ou Gemini CLI
2. **Installe-le** sur ta machine (chez toi *et* à l'école si possible)
3. **Crée un compte gratuit** auprès de l'éditeur correspondant
4. **Configure** l'authentification de l'agent avec ton compte
5. **Vérifie** que l'agent fonctionne : ouvre un dossier vide, lance l'agent, demande-lui de créer un simple fichier `hello.txt`

> 💡 ***Comment trouver les instructions ?*** *Cherche* `"<nom de l'agent> install"` *sur le site officiel de l'éditeur.* ***Pas*** *sur un blog tiers,* ***pas*** *sur YouTube en première intention — la doc officielle est souvent la plus à jour et la plus fiable. Reviens vers les autres sources si tu bloques.*

Partie 2 — Mise en pratique sur un vrai projet

Choisis **un** des scénarios suivants :

- **Scénario A — Démarrer un projet from scratch** : demande à l'agent de créer une petite application (générateur de mot de passe, todo-list, calculatrice, mini-jeu…) dans le langage de ton choix. Suis ses propositions, mais lis chaque diff avant d'accepter.
- **Scénario B — Reprendre un projet existant** : prends un de tes projets en cours (C#, Python, PHP…) et demande à l'agent d'**ajouter une fonctionnalité**. Par exemple, ajouter un bouton *Copier* à ton générateur de mot de passe.
- **Scénario C — Débogage** : prends un code qui ne fonctionne pas et demande à l'agent de l'aider à comprendre pourquoi.

Partie 3 — Observation critique

Pendant que tu utilises l'agent, **note systématiquement** :

- ce qu'il a **bien fait** (tâche réussie du premier coup, suggestion pertinente…)
- ce qu'il a **mal fait** (boucle, hallucination, contournement, oubli d'un détail…)
- les **moments où tu as dû intervenir** pour corriger le tir
- la **différence** par rapport à ce que tu aurais codé toi-même

---

📄 Livrables

À la fin de l'exercice, rends :

1. **Un dépôt git** (local ou GitHub) avec le code produit
2. **Un fichier** `RAPPORT.md` à la racine, contenant :
   - quel agent tu as choisi, et pourquoi
   - les difficultés rencontrées lors de l'installation
   - le scénario choisi (A, B ou C)

- **3 réussites** marquantes de l'agent
- **3 échecs** marquants de l'agent
- ta réponse à la question : *« utiliserai-je cet outil au quotidien ? Dans quelles conditions ? »*

🧪 Bonus — Faire tourner un agent en local (sans compte cloud)

Tu peux pousser l'expérience plus loin : faire tourner Claude Code (ou un agent compatible) sur ta machine **sans envoyer une seule requête sur internet**, en utilisant un modèle de langage qui tourne en local.

Pourquoi c'est intéressant

- **Confidentialité** — ton code ne quitte jamais ta machine
- **Pas de quota** — tu utilises tes propres ressources GPU/CPU
- **Pas de coût** — gratuit même en cas d'usage intensif
- **Ça marche hors ligne** — utile en classe sans wifi fiable

Ce qu'il te faut

ÉLÉMENT	RÔLE
LM Studio	Application desktop qui télécharge et exécute des modèles open source en local. Expose une API compatible OpenAI.
Un modèle de code	Le « cerveau ». Quelques bons choix open source : **Qwen2.5-Coder**, **DeepSeek-Coder-V2**, **Gemma 3**, **Codestral**.
Claude Code (ou autre agent compatible)	Le client qui te permet d'interagir avec le modèle dans le terminal, redirigé vers LM Studio au lieu du cloud.

> ⚠️ ***Limite matérielle réelle*** *: un modèle de code performant demande en général* ***au moins 16 Go de RAM*** *(et idéalement un GPU avec 8+ Go de VRAM). Sur un PC scolaire modeste, prends un modèle plus léger (Qwen 7B, par exemple) — quitte à perdre en qualité. Si ta machine rame trop, repli sur la version cloud avec un compte gratuit.*

La marche à suivre

1. Installe **LM Studio**
2. Télécharge un modèle de code via l'interface de LM Studio (commence par un 7B pour tester)
3. Lance le **serveur local** dans LM Studio (onglet *Developer* → *Start Server*). Note l'URL — par défaut `http://localhost:1234/v1`.
4. Configure ton agent (Claude Code, par exemple) pour qu'il utilise cette URL au lieu de l'API cloud. La doc de l'agent t'expliquera comment changer l'**endpoint** et la **clé API** (mets une chaîne bidon, LM Studio ne vérifie pas).
5. Lance l'agent et utilise-le normalement. Tu verras dans LM Studio les requêtes arriver en direct.

▸ Vidéo de démonstration

La vidéo en haut de cet article (épinglée) montre cette mise en place pas à pas. **Regarde-la avant de te lancer** si tu choisis ce bonus — ça t'évitera plusieurs heures d'errance.

🚨 Règles à respecter pendant l'exercice

RÈGLE	POURQUOI
Lis chaque modification avant de l'accepter	Tu dois rester capable d'expliquer ton code
Tu restes responsable du résultat	Un bug n'est pas la faute de l'IA, c'est la tienne
Pas de données sensibles dans les prompts	Mots de passe, données perso, code propriétaire de stage…
Cite l'IA si tu rends du code généré	Honnêteté intellectuelle
Compte gratuit uniquement à l'école	Règle de l'établissement

✅ Critères de réussite

- ✅ Un agent est installé et fonctionnel sur ta machine
- ✅ Tu sais expliquer **ce qui s'est passé** à chaque étape de ton projet
- ✅ Le code produit fonctionne et a été **relu**
- ✅ Le rapport est honnête : il signale **autant les échecs que les réussites** de l'agent
- ✅ Tu peux montrer en classe une démo de 2-3 minutes

🤔 Questions de réflexion

À ajouter à ton rapport, en quelques phrases chacune :

1. Un agent IA peut écrire du code à ta place. **Qu'est-ce qui reste à ta charge** d'irremplaçable ?
2. Si tu utilisais cet outil pendant tes études, **quels risques** vois-tu pour ton apprentissage ?
3. Est-ce qu'un agent local (LM Studio) et un agent cloud (Claude Code en ligne) **résolvent le même problème** ? Quelles sont les différences pratiques au quotidien ?
4. Imagine que tu sois embauché dans une boîte qui interdit ces outils. **Quels arguments** comprends-tu derrière cette interdiction ?

✍️ À retenir

- Un **agent IA de code** n'écrit pas juste du texte : il **agit** dans ton projet (fichiers, terminal, git).

- Les trois grands gratuits à connaître : **Claude Code**, **Codex CLI**, **Gemini CLI**.
- À l'école : **comptes gratuits uniquement**.
- Un agent peut aussi tourner **100 % en local** via **LM Studio** + un modèle open source (Qwen2.5-Coder, DeepSeek-Coder-V2, Gemma 3…). Confidentiel, sans quota, mais demande une bonne machine.
- **Tu restes responsable** de ce que tu rends — l'agent est un outil, pas un prétexte.



INTERNET

© 2025 Mr. Lambrechts

Internet est un **réseau mondial d'ordinateurs connectés entre eux**. Il permet aux utilisateurs d'accéder et de partager des informations sur une échelle sans précédent. Via Internet, vous pouvez envoyer des emails, consulter des sites web, télécharger des fichiers, et bien plus encore, en quelques clics.

Création de pages Web

introduction

Javascript

introduction

JavaScript - Boucles for - Exercices

> *Les boucles sont des structures de contrôle essentielles en programmation, permettant d'exécuter un bloc de code plusieurs fois. En JavaScript, la boucle `for` est une des plus couramment utilisées. Cet article explore les boucles `for` en JavaScript moderne, avec des explications claires et des exemples concrets.*

Exercices sur les Boucles `for` en JavaScript

Voici 15 exercices de base sur les boucles `for` en JavaScript pour vous aider à vous familiariser avec ce concept essentiel.

Afficher des Nombres de 1 à 10

Utilisez une boucle `for` pour afficher les nombres de 1 à 10 dans la console.

Exemple :

```
for (let i = 1; i <= 10; i++) {
    console.log(i);
}
```

Afficher des Nombres Pairs de 2 à 20

Utilisez une boucle `for` pour afficher les nombres pairs de 2 à 20 dans la console.

Afficher les Éléments d'un Tableau

Déclarez un tableau `noms` contenant les valeurs ["Alice", "Bob", "Charlie"], puis utilisez une boucle `for` pour afficher chaque nom dans la console.

Somme des Nombres de 1 à 10

Utilisez une boucle `for` pour calculer et afficher la somme des nombres de 1 à 10.

Afficher une Table de Multiplication

Utilisez une boucle `for` pour afficher la table de multiplication de 5 (5, 10, 15, ..., 50).

Afficher des Nombres Impairs de 1 à 19

Utilisez une boucle `for` pour afficher les nombres impairs de 1 à 19.

Afficher les Nombres en Ordre Inverse

Utilisez une boucle `for` pour afficher les nombres de 10 à 1 dans la console.

▸ Calculer la Factorielle d'un Nombre

Utilisez une boucle `for` pour calculer la factorielle de 5 et afficher le résultat.

La factorielle d'un nombre entier naturel, notée `n!`, est le produit de tous les entiers positifs inférieurs ou égaux à `n` (soit 1x2x3x4...x n). Par exemple, 4! = 1 x 2 x 3 x 4 = 24.

🎯 Bases de Javascript: Carnet d'objectifs

> ***Introduction au développement web interactif** · 4e–6e secondaire · Informatique · ~10h*

NOM	CLASSE	ANNÉE

> ***Légende :*** *☐ = à valider · **Élève** = autoévaluation · **Pair** = évaluation par un(e) camarade · **Prof** = validation de l'enseignant(e)*

A. Environnement & premiers pas en JavaScript

Niveau 1 — Reproduire

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable d'ouvrir la console du navigateur (F12) et d'y exécuter du code JS.	☐	☐	☐	
Je suis capable d'intégrer du JavaScript dans une page HTML via la balise `<script>`.	☐	☐	☐	
Je suis capable d'utiliser `alert()`, `console.log()` et `document.write()` pour afficher un message.	☐	☐	☐	
Je suis capable de distinguer un script interne (inline) d'un script externe (fichier `.js`).	☐	☐	☐	

Niveau 2 — Comprendre

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je sais expliquer le rôle de JavaScript dans le trio HTML / CSS / JS.	☐	☐	☐	
Je sais choisir la méthode d'affichage adaptée selon le contexte (débogage vs utilisateur).	☐	☐	☐	

B. Variables, types et conventions

Niveau 1 — Reproduire

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable de déclarer une variable avec `let` ou `const`.	☐	☐	☐	
Je suis capable d'utiliser les types `string`, `number` et `boolean`.	☐	☐	☐	
Je suis capable d'utiliser les opérateurs arithmétiques et de concaténer des chaînes.	☐	☐	☐	
Je suis capable de recueillir une saisie utilisateur avec `prompt()` et de l'afficher.	☐	☐	☐	

Niveau 2 — Comprendre

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je sais expliquer la différence entre `let` et `const`, et choisir le bon mot-clé.	☐	☐	☐	
Je sais nommer mes variables en camelCase et justifier l'importance des conventions de nommage.	☐	☐	☐	
Je sais convertir une saisie texte en nombre avec `parseInt()` / `parseFloat()` si nécessaire.	☐	☐	☐	

C. Structures conditionnelles

Niveau 1 — Reproduire

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable d'écrire un bloc `if / else if / else` fonctionnel.	☐	☐	☐	
Je suis capable d'utiliser les opérateurs de comparaison (`===`, `!==`, `<`, `>`, `≤`, `≥`).	☐	☐	☐	

Niveau 2 — Comprendre

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je sais combiner plusieurs conditions avec `&&` et `||`.	☐	☐	☐	
Je sais expliquer la différence entre `==` et `===` et utiliser `===` par défaut.	☐	☐	☐	
Je sais concevoir un algorithme de décision simple sous forme de condition imbriquée.	☐	☐	☐	

D. Boucles et fonctions

Niveau 1 — Reproduire

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable d'écrire une boucle `for` et une boucle `while`.	☐	☐	☐	
Je suis capable de déclarer une fonction, lui passer des paramètres et récupérer un `return`.	☐	☐	☐	

Niveau 2 — Comprendre

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je sais choisir entre `for` et `while` selon la situation.	☐	☐	☐	
Je sais décomposer un problème en fonctions réutilisables.	☐	☐	☐	
Je sais identifier et corriger une boucle infinie.	☐	☐	☐	

Niveau 3 — Généraliser / Combiner

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable de concevoir une fonction qui intègre une boucle et des conditions pour résoudre un problème inédit.	☐	☐	☐	

E. Manipulation du DOM & événements

Niveau 1 — Reproduire

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable de sélectionner un élément HTML avec `getElementById` et `querySelector`.	☐	☐	☐	
Je suis capable de modifier `innerHTML`, `textContent` et `style` d'un élément.	☐	☐	☐	
Je suis capable d'associer un événement (`click`, `keyup`) à une fonction via `addEventListener`.	☐	☐	☐	

Niveau 2 — Comprendre

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je sais expliquer ce qu'est le DOM et comment JS interagit avec la page HTML.	☐	☐	☐	
Je sais lire la valeur d'un champ `<input>` et l'utiliser dans mon script.	☐	☐	☐	

Niveau 3 — Généraliser / Combiner

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable de créer une interaction complète : saisie utilisateur → traitement JS → mise à jour de la page, sans modèle fourni.	☐	☐	☐	

F. Structures de données : tableaux, objets et JSON

Niveau 1 — Reproduire

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable de créer un tableau, y ajouter des éléments (`push`) et accéder à un index.	☐	☐	☐	
Je suis capable de créer un objet littéral avec des propriétés et d'y accéder (dot notation).	☐	☐	☐	
Je suis capable de convertir un objet en JSON (`JSON.stringify`) et l'inverse (`JSON.parse`).	☐	☐	☐	

Niveau 2 — Comprendre

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je sais parcourir un tableau avec une boucle `for` ou `forEach` et afficher les résultats dans la page.	☐	☐	☐	
Je sais expliquer la différence entre un tableau et un objet, et choisir la structure adaptée.	☐	☐	☐	
Je sais expliquer l'utilité du format JSON dans les échanges entre applications web.	☐	☐	☐	

Niveau 3 — Généraliser / Combiner

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable de stocker des données structurées (tableau d'objets) et de les afficher dynamiquement dans le DOM.	☐	☐	☐	

G. Projet intégrateur

Niveau 3 — Généraliser / Transférer

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je suis capable de concevoir et réaliser une page web interactive de A à Z (HTML + JS) sur un sujet de mon choix.	☐	☐	☐	
Je suis capable de présenter mon projet, d'expliquer mes choix techniques et de répondre aux questions.	☐	☐	☐	
Je suis capable de lire et comprendre le code d'un(e) camarade et d'y apporter une suggestion constructive.	☐	☐	☐	
Je sais identifier et corriger un bug en utilisant la console du navigateur de façon autonome.	☐	☐	☐	

H. Autonomie & transfert professionnel

Niveau 3 — Généraliser / Transférer

OBJECTIF	ÉLÈVE	PAIR	PROF	RESSOURCE
Je sais rechercher une solution technique de façon autonome (documentation MDN, StackOverflow).	☐	☐	☐	
Je sais structurer mon code de manière lisible : indentation, nommage cohérent, commentaires utiles.	☐	☐	☐	
Je sais situer JavaScript dans le paysage professionnel du développement web (front-end, frameworks, métiers).	☐	☐	☐	
Je suis capable d'adapter un script existant à un nouveau besoin sans partir de zéro.	☐	☐	☐	

JavaScript - La console développeur

> *Avant d'écrire des scripts, il faut savoir où **les exécuter** et où **lire leurs messages**. La console développeur, intégrée à tous les navigateurs, est l'outil n°1 du développeur web : essai rapide de code, affichage de variables, traque des erreurs.*

Ouvrir la console

NAVIGATEUR	RACCOURCI
Chrome / Edge	`F12` ou `Ctrl+Shift+I`
Firefox	`F12` ou `Ctrl+Shift+K`
Safari	`Cmd+Option+C` (activer d'abord le menu Développement dans les préférences)

Un panneau apparaît en bas (ou sur le côté) de la page. Plusieurs onglets : **Console**, **Éléments**, **Réseau**, **Sources**, etc. On reste sur **Console** pour commencer.

Exécuter du JavaScript à la volée

Le prompt de la console accepte du JavaScript. Tape, valide avec Entrée — c'est exécuté immédiatement dans le contexte de la page courante.

```
> 2 + 2
4

> let nom = "Alice"
undefined

> `Bonjour ${nom}`
"Bonjour Alice"

> document.title
"Mon super site"
```

C'est parfait pour :

- tester une expression rapidement,
- explorer un objet (`document` , `window` …),
- corriger une page en direct (modifier le texte, le style…) — les modifications sont perdues au rechargement, c'est juste pour expérimenter.

`console.log()` — afficher depuis ton script

C'est l'outil de débogage le plus utilisé au monde, tous langages confondus.

```
let age = 17;
let nom = "Alice";

console.log(nom);                       // "Alice"
console.log(age);                       // 17
console.log("Nom :", nom, "| Âge :", age);   // Nom : Alice | Âge : 17
console.log({ nom, age });              // { nom: "Alice", age: 17 }
```

Quelques astuces :

- Passe **plusieurs arguments** séparés par des virgules — la console les affiche les uns à la suite des autres.
- Pour visualiser un objet de façon lisible, passe-le directement : `console.log(monObjet)` affiche un arbre dépliable.

▸ `console.warn` , `console.error` , `console.table`

Variantes utiles selon le contexte :

```
console.warn("Attention, valeur étrange");   // jaune, avec icône avertissement
console.error("Quelque chose a planté");     // rouge, avec stack trace
console.table([
    { nom: "Alice", age: 16 },
    { nom: "Bob",   age: 17 }
]);                                          // tableau visuel
```

`console.table` sur un tableau d'objets est particulièrement spectaculaire — essaie une fois et tu l'utiliseras tout le temps.

Lire les erreurs

Quand ton script plante, le navigateur affiche une **erreur rouge** dans la console. Elle contient en général :

1. Le **message** d'erreur (`Uncaught ReferenceError: foo is not defined`).
2. Le **fichier** et la **ligne** où ça plante (cliquable !).
3. La **stack trace** (la chaîne d'appels qui a mené à l'erreur).

Quelques erreurs classiques :

MESSAGE	CAUSE TYPIQUE
`Uncaught ReferenceError: x is not defined`	Variable mal orthographiée ou non déclarée
`Uncaught TypeError: ... is null`	Tu utilises un élément avant qu'il existe (script avant `<body>`)
`Uncaught SyntaxError: Unexpected token`	Erreur de syntaxe : parenthèse, accolade, virgule manquante
`is not a function`	Tu appelles `()` sur quelque chose qui n'est pas une fonction (faute de frappe sur le nom)

Lire l'erreur, c'est 80% du débogage. Beaucoup de débutants paniquent à la première erreur rouge et la ferment. Au contraire : la console te dit **exactement** ce qui cloche et **où**. Lis le message, clique sur le lien de la ligne, et la moitié du problème est déjà résolu.

`alert`, `console.log`, `document.write` — lequel ?

Trois manières d'afficher quelque chose. Elles ne sont **pas équivalentes**.

MÉTHODE	POUR QUOI ?
`alert("msg")`	Message à l'**utilisateur** (boîte de dialogue bloquante). À utiliser parcimonieusement.
`console.log(...)`	Message **pour toi**, le développeur. À utiliser sans modération.
`document.write(...)`	Insère dans la page. **À éviter** : casse complètement la page si appelé après chargement.

Règle simple : `console.log` **pour déboguer,** `alert` **pour parler à l'utilisateur, jamais** `document.write`.

Inspecter un élément

Au-delà de la console, l'onglet **Éléments** (ou **Inspecteur** sur Firefox) montre l'arbre HTML de la page **en direct**.

- Clic droit → *Inspecter* sur n'importe quel élément de la page → l'inspecteur s'ouvre sur cet élément.
- Tu peux modifier les attributs, le texte, le CSS en temps réel (les changements sont éphémères).
- Tu peux suivre les modifications faites par ton JS : très utile pour vérifier que ton script change bien ce que tu crois.

À retenir

- `F12` ouvre la console.
- Le panneau **Console** exécute du JS dans le contexte de la page : un mini-laboratoire.

- `console.log( ... )` est ton meilleur ami pour comprendre ce qu'il se passe dans ton script. Sème-en partout pendant le débogage, retire-les avant la version finale.
- Les erreurs rouges ne sont pas un échec : elles te disent **où** et **quoi** corriger. Lis-les.
- `alert` pour l'utilisateur, `console.log` pour toi, jamais `document.write`.

JavaScript - Syntaxe et variables

> *JavaScript est un langage incontournable du Web moderne. Cette page présente l'essentiel de sa syntaxe afin de pouvoir écrire rapidement des scripts simples, lisibles et efficaces.*

Variables

En JavaScript, une variable permet de **stocker une valeur en mémoire** afin de la réutiliser ou l'afficher plus tard dans le programme.

Aujourd'hui, pour déclarer une variable, on utilise presque exclusivement `let` et `const`.

```
let age = 30;
let nom = "Jean";
const PI = 3.14;
```

- `let` : variable **modifiable**, portée limitée au bloc `{}`
- `const` : variable **non modifiable** (constante)
- `var` : ancienne syntaxe, **à éviter** (problèmes de portée et de lisibilité)

👉 **Bonne pratique** :

- utiliser `const` par défaut
- utiliser `let` uniquement si la valeur doit changer
- Tu verras `var` dans d'anciens tutoriels. Il fonctionne, mais il a des comportements imprévisibles. **On ne l'utilise plus.**

Types de données

JavaScript est un langage **à typage dynamique** : le type est déterminé automatiquement lors de l'affectation d'une valeur.

```
let nombre = 10;                // Number
let pi = 3.1415;                // Number
let prenom = "Alice";           // String
let estConnecte = true;         // Boolean
let fruits = ["pomme", "banane"]; // Array
let personne = { nom: "Jean", age: 30 }; // Object
```

Types courants : `Number`, `String`, `Boolean`

`string` — chaîne de caractères

Du texte, toujours entre guillemets (simples, doubles, ou backticks).

```
let ville = "Bruxelles";
let message = 'Bonjour tout le monde';
let phrase = `La ville est ${ville}`; // template literal
```

Les backticks (`` ` ``) permettent d'insérer des variables directement dans le texte avec `${}` — c'est très pratique.

```
let prenom = "Lucas";
let age = 16;
console.log(`Je m'appelle ${prenom} et j'ai ${age} ans.`);
// → Je m'appelle Lucas et j'ai 16 ans.
```

`number` — nombre

Entiers et décimaux, sans guillemets.

```
let temperature = -3;
let prix = 9.99;
let annee = 2025;
```

`boolean` — vrai ou faux

Exactement deux valeurs possibles : `true` ou `false`.

```
let estConnecte = true;
let aPayé = false;
```

Les booléens sont au cœur de toutes les conditions et décisions du programme.

Conventions de nommage

Le code qu'on écrit est lu par d'autres humains (et par son futur soi). Un bon nommage, ça évite des heures de confusion.

CAMELCASE — LA CONVENTION JS

En JavaScript, les variables et fonctions s'écrivent en **camelCase** : la première lettre est minuscule, chaque mot suivant commence par une majuscule.

```
// ✅ Correct
let nombreDEleves = 28;
let prixUnitaire = 4.5;
let nomUtilisateur = "Amir";

// ❌ À éviter
let nombre_d_eleves = 28;  // snake_case (Python, pas JS)
let PrixUnitaire = 4.5;    // PascalCase (réservé aux classes)
let p = 4.5;               // trop court, sans sens
```

AUTRES RÈGLES

- **Nom descriptif** : `scoreJoueur` plutôt que `sj` ou `x`
- **Commence par une lettre** : pas de chiffre en début (`3prix` → ❌)
- **Pas d'accents ni d'espaces** : `nombreEtudiants` et non `nombreÉtudiants`
- **Respecte la casse** : `age` , `Age` et `AGE` sont trois variables différentes

```
// Exemple concret bien nommé
const nombreDeVies = 3;
let scoreActuel = 0;
let nomDuJoueur = prompt("Ton pseudo ?");

console.log(`${nomDuJoueur} commence avec ${nombreDeVies} vies.`);
```

> *Un bon nom de variable, c'est un commentaire gratuit. Si tu dois ajouter un commentaire pour expliquer ce que contient une variable, c'est souvent signe que son nom n'est pas assez clair.*

À retenir

- `const` par défaut, `let` si la valeur change. Jamais `var` .
- Les trois types de base : `string` (texte), `number` (nombre), `boolean` (vrai/faux).
- `prompt()` renvoie toujours un `string` → pense à `parseInt()` ou `parseFloat()` si tu fais des calculs.
- Les backticks + `${}` rendent les messages beaucoup plus lisibles que la concaténation avec `+` .
- **camelCase** pour tous les noms de variables. Toujours descriptif, jamais `x` ou `temp` .

Opérateurs arithmétiques

Les opérateurs permettent de **manipuler les valeurs numériques**.

OPÉRATEUR	SYMBOLE	DESCRIPTION	EXEMPLE	RÉSULTAT
Addition	`+`	Additionne deux valeurs	`5 + 3`	`8`
Soustraction	`-`	Soustrait une valeur	`10 - 4`	`6`
Multiplication	`*`	Multiplie deux valeurs	`6 * 2`	`12`
Exposant	`**`	Exposant	`2 ** 2`	`4`
Division	`/`	Division classique	`15 / 3`	`5`
Modulo	`%`	Reste de la division entière	`10 % 3`	`1`
Incrémentation	`++`	Ajoute 1	`x++`	
Décrémentation	`--`	Retire 1	`x--`	

Exemples:

```
let a = 10;
let b = 3;

console.log(a + b);  // → 13  (addition)
console.log(a - b);  // → 7   (soustraction)
console.log(a * b);  // → 30  (multiplication)
console.log(a / b);  // → 3.333...  (division)
console.log(a % b);  // → 1   (modulo : reste de la division)
console.log(a ** b); // → 1000  (puissance)
```

Le **modulo** (`%`) est très utile en pratique : il permet de savoir si un nombre est pair, de créer des cycles, etc.

```
console.log(8 % 2);  // → 0  (pair)
console.log(7 % 2);  // → 1  (impair)
```

▸ PEMDAS

JavaScript respecte strictement la règle PEMDAS, comme la plupart des langages de programmation.

PEMDAS définit l'ordre de priorité des opérations arithmétiques :

- Parentheses → ()
- Exposants → **
- Multiplication → *
- Division → /
- Addition → +
- Subtraction → -

Si 2 opérations ont la même priorité, elles sont exécutées de gauche à droite.

```
let resultat = 20 / 5 * 2;
console.log(resultat);
```

Résultat: `8`. Division et multiplication ont la même priorité, l'évaluation se fait de gauche à droite.

▸ Division entière

Contrairement à Python, JavaScript ne possède pas d'opérateur de division entière. On utilise donc `Math.floor()`.

```
let resultat = Math.floor(10 / 3); // 3
```

Points-virgules

Les points-virgules `;` marquent la fin d'une instruction.

JavaScript peut parfois s'en passer, mais **il est fortement recommandé de les utiliser**.

Toujours utiliser les points-virgules pour éviter les erreurs difficiles à détecter.

```
let x = 5;
let y = 10;
```

Conclusion

Cette page couvre les **fondations indispensables** de JavaScript : variables, types, opérateurs et structure du code. Ces bases sont importantes avant d'aborder les conditions, les boucles, les fonctions et la programmation Web dynamique.

JavaScript - Exercices de base

> *Voici **10 exercices courts, ciblés et cohérents pour une introduction**, exclusivement centrés sur **la déclaration de variables** et **les calculs simples**. Ils sont progressifs, sans fonctions, sans conditions, sans boucles, et parfaitement adaptés à un **premier contact avec JavaScript**.*

Pour afficher une variable, tu peux utiliser la fonction `alert()` ou `console.log()`.

```
let resultat = 5 + 2 * 10;
alert(resultat);
console.log(resultat);
```

Exercice 1: Déclaration simple

Déclare une variable `age` et affecte-lui la valeur `16`. Affiche ensuite la valeur dans une alerte.

Exercice 2: Deux variables

Déclare deux variables `nombre1` et `nombre2` contenant des nombres (avec des valeurs de ton choix). Affiche leur somme.

Ensuite, affiche dans la console leur somme, leur différence, leur produit et leur quotient, avec un libellé clair pour chaque résultat.

Exercice 3: Calcul intermédiaire

Déclare une variable `prix` valant `12.5` et une variable `quantite` valant `4`. Calcule le prix total et stocke-le dans une variable `total`.

Exercice 3bis: Calcul intermédiaire Pareil mais en utilisant `prompt()` 😉

Exercice 4: Modification de variable

Déclare une variable `score` avec la valeur `10`. Augmente-la de `5` et affiche le résultat.

Exercice 5: Incrémentation

Déclare une variable `compteur` valant `0`. Incrémente-la deux fois et affiche sa valeur finale.

Exercice 6: Moyenne simple

Déclare trois variables `note1`, `note2`, `note3`. Calcule la moyenne et stocke-la dans une variable `moyenne`.

Exercice 7: Modulo

Déclare une variable `nombre` contenant un entier. Calcule le reste de la division par `2` et stocke le résultat dans une variable `reste`.

Exercice 8: Aire d'un rectangle

Déclare deux variables `largeur` et `hauteur`. Calcule l'aire du rectangle et stocke-la dans une variable `aire`.

Exercice 9: Conversion simple

Déclare une variable `celsius` contenant une température. Calcule la température correspondante en Fahrenheit (`F = C × 1.8 + 32`).

Exercice 10: Priorité des opérations

Déclare trois variables `a` , `b` et `c` . Calcule le résultat de l'expression suivante et stocke-le dans une variable `resultat` :

```
a + b * c
```

Exercice 11: Présentation automatique Demande à l'utilisateur son prénom et son année de naissance. Calcule son âge approximatif et affiche une phrase complète dans la console.

```
// Exemple de sortie attendue :
// "Bonjour Léa ! Tu as environ 17 ans."
```

JavaScript - La saisie utilisateur

> *Pour qu'un programme soit interactif, il doit pouvoir **lire ce que tape l'utilisateur**. `prompt()` est le moyen le plus simple — mais attention au piège classique : ce qu'il renvoie n'est jamais un nombre, même quand on tape un chiffre.*

`prompt()` — demander une valeur

`prompt()` ouvre une petite boîte de dialogue avec un champ de saisie. Quand l'utilisateur valide, la fonction renvoie ce qu'il a tapé.

```
let nom = prompt("Quel est ton prénom ?");
console.log(`Bonjour ${nom}`);
```

On peut aussi proposer une **valeur par défaut** en deuxième argument :

```
let pseudo = prompt("Ton pseudo ?", "Anonyme");
```

Si l'utilisateur clique sur **Annuler**, `prompt()` renvoie `null`. On peut tester :

```
let nom = prompt("Ton prénom ?");

if (nom === null) {
    console.log("Saisie annulée");
} else if (nom === "") {
    console.log("Saisie vide");
} else {
    console.log(`Bonjour ${nom}`);
}
```

Le piège : prompt renvoie toujours un texte

C'est **le bug** qui surprend tout le monde au début.

```
let nombre = prompt("Donne un nombre :");
// L'utilisateur tape : 5

console.log(nombre + 3);
// Affiche : "53"  (et non 8)
```

Pourquoi ? Parce que `nombre` contient **le texte** `"5"`, pas le nombre 5. Et avec l'opérateur `+`, JavaScript préfère **concaténer** quand au moins un côté est du texte : `"5" + 3` donne `"53"`.

`prompt()` **renvoie *toujours* un string**, même quand l'utilisateur tape un chiffre. Si tu veux calculer avec, tu **dois** convertir.

Convertir en nombre

Trois outils, à choisir selon le contexte.

`parseInt()` — nombre entier

Lit le texte caractère par caractère jusqu'au premier qui n'est pas un chiffre. Idéal pour les âges, scores, quantités…

```
let age = parseInt(prompt("Ton âge ?"));
// "16"   → 16
// "16.7" → 16     (la partie décimale est ignorée)
// "16ab" → 16     (s'arrête au premier non-chiffre)
// "abc"  → NaN    (Not a Number)
```

`parseFloat()` — nombre décimal

Pareil mais accepte le point décimal.

```
let prix = parseFloat(prompt("Prix en euros ?"));
// "9.99"  → 9.99
// "9,99"  → 9        (la virgule arrête la lecture — attention !)
// "abc"   → NaN
```

Virgule ou point ? JavaScript utilise le **point** comme séparateur décimal, comme dans la plupart des langages de programmation. Tes utilisateurs francophones taperont souvent une virgule. Tu peux convertir : `texte.replace(",", ".")` avant le `parseFloat`.

`Number()` — strict

Plus exigeant : il faut que **tout le texte** soit un nombre, sinon `NaN`.

```
Number("16")      // 16
Number("16.7")    // 16.7
Number("16ab")    // NaN   (≠ parseInt, qui aurait renvoyé 16)
Number("")        // 0     (cas un peu surprenant)
```

Pour la plupart des cas pédagogiques, `parseInt` et `parseFloat` suffisent.

NaN — le résultat des conversions ratées

Quand on essaie de convertir un texte qui n'a rien de numérique, on obtient `NaN` (*Not a Number*). C'est techniquement un nombre, mais qui propage l'erreur dans tous les calculs.

```
let age = parseInt("salut");
console.log(age);          // NaN
console.log(age + 10);     // NaN
console.log(age * 2);      // NaN
```

On teste avec `isNaN()` :

```
let age = parseInt(prompt("Ton âge ?"));

if (isNaN(age)) {
    alert("Ce n'est pas un nombre valide.");
} else {
    alert(`Tu as ${age} ans.`);
}
```

Bonne pratique : **toujours vérifier** que la conversion a marché avant d'utiliser le résultat.

Pattern complet : saisie sécurisée

Le pattern classique combine prompt + conversion + validation :

```
let age = parseInt(prompt("Ton âge ?"));

if (isNaN(age) || age < 0 || age > 120) {
    alert("Âge invalide.");
} else {
    alert(`Tu as ${age} ans.`);
}
```

Si tu veux **boucler tant que la saisie n'est pas valide**, c'est typiquement le rôle d'un `while` :

```
let age;

do {
    age = parseInt(prompt("Ton âge ?"));
} while (isNaN(age) || age < 0 || age > 120);

alert(`Tu as ${age} ans.`);
```

Autres méthodes : `alert` et `confirm`

`prompt` a deux cousines plus simples.

▸ `alert()` — afficher un message

```
alert("Bienvenue !");
```

Ne renvoie rien d'utile. Sert juste à afficher.

▸ `confirm()` — demander oui/non

```
let veutContinuer = confirm("Veux-tu continuer ?");

if (veutContinuer) {
    console.log("OK, on continue");
} else {
    console.log("Abandon");
}
```

Renvoie `true` si OK, `false` si Annuler. Pratique pour les questions binaires.

Limites de `prompt`

`prompt()` est parfait pour apprendre, mais en pratique, on l'utilise rarement sur un vrai site :

- **Apparence figée** — impossible à styler avec du CSS.
- **Bloquant** — la page se gèle tant que l'utilisateur n'a pas répondu.
- **Mauvais pour mobile** — l'expérience est dégradée.

Sur un vrai site, on utilise des **champs** `<input>` dans un formulaire HTML. On lit leur valeur avec `document.getElementById("monChamp").value`. Mais ça, c'est l'étape suivante.

À retenir

- `prompt("question")` ouvre une boîte de saisie et renvoie ce que tape l'utilisateur.
- Le résultat est **toujours un string**, même `"42"` n'est pas le nombre 42.
- `parseInt(texte)` pour un entier, `parseFloat(texte)` pour un décimal.
- `isNaN(valeur)` vérifie si la conversion a échoué — toujours tester avant d'utiliser.
- `alert("message")` pour afficher, `confirm("question")` pour oui/non.
- `prompt` est un outil d'apprentissage. Sur un vrai site : des champs `<input>` HTML.

JavaScript - Les conditions

> *Un programme ne se contente pas d'exécuter des instructions dans l'ordre. Il doit aussi pouvoir **prendre des décisions** : l'utilisateur est-il majeur ? le score dépasse-t-il 100 ? le mot de passe est-il correct ? En JavaScript, ces décisions s'écrivent avec `if`, `else` et `else if`.*

Une condition, c'est une question

Une condition est une **question à laquelle on peut répondre par vrai ou faux** :

- L'âge est-il supérieur à 18 ?
- Le mot de passe est-il correct ?
- Le score dépasse-t-il 100 ?

Selon la réponse, le programme suit un chemin ou un autre.

`if` — exécuter du code seulement si...

La forme la plus simple : un seul bloc, exécuté **uniquement si** la condition est vraie.

```
if (condition) {
    // instructions exécutées si la condition est vraie
}
```

Exemple :

```
let age = 20;

if (age >= 18) {
    console.log("Vous êtes majeur");
}
```

Si la condition est fausse, **rien ne se passe** — le programme continue sans exécuter le bloc.

`if / else` — deux chemins, un seul emprunté

Pour exécuter **un bloc si vrai, un autre si faux** :

```
let age = 16;

if (age >= 18) {
    console.log("Accès autorisé");
} else {
    console.log("Accès refusé");
}
```

Une seule des deux branches est exécutée. Jamais les deux, jamais aucune.

```
       condition ?
       /         \
     vrai        faux
      |           |
   bloc A      bloc B
```

`if / else if / else` — plusieurs cas

Quand il y a **plus de deux possibilités**, on enchaîne des `else if`.

```
let note = 72;

if (note >= 90) {
    console.log("Excellent");
} else if (note >= 75) {
    console.log("Très bien");
} else if (note >= 50) {
    console.log("Réussi");
} else {
    console.log("Échec");
}
```

Les conditions sont testées **dans l'ordre**. Dès qu'une est vraie, son bloc s'exécute et **les suivantes ne sont plus évaluées**.

Le `else` final attrape tout ce qui n'a été capturé par aucune condition précédente. Il est optionnel.

Les opérateurs de comparaison

OPÉRATEUR	SIGNIFICATION
`>`	supérieur à
`<`	inférieur à
`≥`	supérieur ou égal
`≤`	inférieur ou égal
`===`	égal
`≠==`	différent

Utilise `===` (et `≠==`), pas `==` (et `≠`).

`==` compare en convertissant les types : `"5" == 5` est `true` (bizarre). `===` compare sans conversion : `"5" === 5` est `false` (logique). Prends l'habitude de `===` dès maintenant, tu t'épargneras des heures de bugs.

L'ordre des conditions compte

Quand on teste des **seuils** (notes, scores, âges…), l'ordre n'est pas neutre. JavaScript s'arrête à la première condition vraie.

```
// ❌ Ordre incorrect
let note = 80;

if (note >= 50) {
    console.log("Réussi");        // ← s'affiche ici
} else if (note >= 75) {
    console.log("Très bien");     // jamais atteint
}
```

Note = 80 est ≥ 50, donc on affiche "Réussi" et c'est fini. "Très bien" n'est jamais testé.

```
// ✅ Du plus exigeant au plus permissif
if (note >= 75) {
    console.log("Très bien");
} else if (note >= 50) {
    console.log("Réussi");
}
```

Règle pratique :

- comparaisons `≥` ou `>` → tester du **plus grand au plus petit**

- comparaisons `≤` ou `<` → tester du **plus petit au plus grand**

Exemple complet : saisie utilisateur

```
let age = prompt("Quel est votre âge ?");
age = parseInt(age);   // prompt() renvoie un string, on le convertit en nombre

if (age >= 18) {
    alert("Bienvenue !");
} else if (age >= 12) {
    alert("Accès restreint.");
} else {
    alert("Trop jeune.");
}
```

`prompt()` renvoie **toujours un texte**, même quand on tape un nombre. `parseInt()` est obligatoire avant toute comparaison numérique.

Erreurs fréquentes

Confondre `=` et `===`

```
if (age = 18) { ... }    // ❌ assigne 18 à age, condition toujours "vraie"
if (age === 18) { ... }  // ✅ compare age à 18
```

C'est la cause de bug la plus classique chez les débutants. `==` toujours, sauf si tu sais ce que tu fais.

Oublier les accolades

```
// ❌ Trompeur
if (age >= 18)
    console.log("Bienvenue");
    console.log("Tu peux entrer");   // ← TOUJOURS exécuté, indépendamment du if
```

Sans accolades, **seule la première instruction** dépend du `if`. L'indentation ne change rien — c'est un piège visuel. Mets toujours les `{ }`.

Oublier les parenthèses

```
if age >= 18 { ... }    // ❌ erreur de syntaxe
if (age >= 18) { ... }  // ✅
```

À retenir

- `if (condition) { ... }` — exécute si vrai, ignore sinon.
- `if / else` — un chemin sur deux.
- `if / else if / else` — plusieurs cas, **dans l'ordre**, seule la première condition vraie est exécutée.
- `===` toujours, jamais `==`.
- Toujours les `{ }`, même pour une seule instruction.
- Pour les seuils : ordre **décroissant** avec `≥`, **croissant** avec `≤`.

Pour combiner plusieurs conditions dans un seul test (ET, OU), voir l'article suivant.

JavaScript - Combiner plusieurs conditions

> *Une seule condition ne suffit pas toujours : un utilisateur doit être majeur **et** avoir un billet, un joueur gagne avec un gros score **ou** un bonus. Les opérateurs logiques `&&`, `||` et `!` permettent de combiner plusieurs tests dans une seule condition.*

Les trois opérateurs logiques

OPÉRATEUR	LECTURE	VRAI QUAND…
`&&`	**ET**	les **deux** conditions sont vraies
`||`	**OU**	**au moins une** condition est vraie
`!`	**NON**	la condition est **fausse** (négation)

`&&` — ET (les deux à la fois)

```
let age = 20;
let aBillet = true;

if (age >= 18 && aBillet) {
    console.log("Entrée autorisée");
}
```

Lecture : *"si l'âge est ≥ 18 **et** que la personne a un billet, alors..."*

Si **une seule** des deux conditions est fausse, le tout est faux.

```
   condition 1   ET   condition 2
       vrai            vrai
         \              /
          résultat : vrai
```

`||` — OU (au moins une)

```
let role = "admin";
let age = 17;

if (role === "admin" || age >= 18) {
    console.log("Accès autorisé");
}
```

Lecture : *"si l'utilisateur est admin **ou** qu'il est majeur..."*

Il suffit qu'**une seule** condition soit vraie. Les deux peuvent l'être aussi, ça reste vrai.

`!` — NON (négation)

`!` inverse une valeur booléenne : `!true` devient `false`, et inversement.

```
let estConnecte = false;

if (!estConnecte) {
    console.log("Veuillez vous connecter");
}
```

Lecture : *"si **non** connecté..."*

`!` se lit souvent comme *"pas"* ou *"non"* devant la variable.

Combiner plusieurs opérateurs

On peut chaîner autant d'opérateurs que nécessaire. Les **parenthèses** rendent la logique explicite :

```
let age = 20;
let aBillet = true;
let estVIP = false;

if ((age >= 18 && aBillet) || estVIP) {
    console.log("Entrée autorisée");
}
```

Lecture : *"un VIP entre toujours ; sinon il faut être majeur **et** avoir un billet."*

Conseil : dès qu'on combine `&&` et `||`, **mets des parenthèses**. Sans elles, `&&` a priorité sur `||` (comme `*` sur `+` en maths) — c'est une source classique de bugs. Les parenthèses ne coûtent rien et rendent l'intention évidente.

L'évaluation est paresseuse

JavaScript évalue les conditions **de gauche à droite** et s'arrête dès qu'il connaît le résultat :

```
if (utilisateur && utilisateur.nom === "Alice") {
    // ...
}
```

Si `utilisateur` est `null` ou `undefined`, la première partie est fausse, et **la deuxième n'est pas évaluée**. Sans cela, on aurait une erreur en lisant `.nom` sur `null`.

C'est une astuce courante : tester l'existence **avant** d'utiliser une propriété.

Exemples concrets

Contrôle d'accès

```
let role = "membre";
let estActif = true;
let estBanni = false;

if ((role === "admin" || role === "membre") && estActif && !estBanni) {
    console.log("Accès autorisé");
}
```

Validation de formulaire

```
let email = "alice@example.com";
let motDePasse = "secret123";

if (email !== "" && motDePasse.length >= 8) {
    console.log("Formulaire valide");
} else {
    console.log("Champs manquants ou trop courts");
}
```

Jeu : bonus

```
let score = 150;
let aPotion = true;
let aClef = false;

// On accède au boss si on a un gros score, OU si on a une potion ET une clef
if (score >= 200 || (aPotion && aClef)) {
    console.log("Boss débloqué !");
}
```

À retenir

- `&&` = ET — vrai si **toutes** les conditions sont vraies.
- `||` = OU — vrai si **au moins une** condition est vraie.
- `!` = NON — inverse vrai/faux.
- Quand on mélange `&&` et `||`, **toujours des parenthèses**.
- L'évaluation s'arrête dès que le résultat est connu (évaluation paresseuse). Utile pour tester l'existence d'une variable avant de l'utiliser.

JavaScript - La boucle for

> *Une boucle exécute un bloc de code **plusieurs fois**. La boucle `for` est utilisée quand on sait combien de fois on veut répéter — afficher les 10 premiers nombres, parcourir tous les éléments d'un tableau, dessiner une grille de 8×8 cases.*

La syntaxe en trois parties

```
for (initialisation; condition; incrément) {
    // instructions répétées
}
```

Les trois parties séparées par des `;` :

- **initialisation** — exécutée **une seule fois**, au début. Souvent une déclaration de compteur.
- **condition** — vérifiée **avant chaque tour**. Tant qu'elle est vraie, la boucle continue. Si fausse, la boucle s'arrête.
- **incrément** — exécuté **après chaque tour**, généralement pour faire avancer le compteur.

```
for (let i = 1; i <= 5; i++) {
    console.log(i);
}
// affiche : 1 2 3 4 5
```

Déroulement :

```
i = 1   → 1 ≤ 5 ? oui → console.log(1) → i devient 2
i = 2   → 2 ≤ 5 ? oui → console.log(2) → i devient 3
 ...
i = 5   → 5 ≤ 5 ? oui → console.log(5) → i devient 6
i = 6   → 6 ≤ 5 ? non → fin de la boucle
```

Variations courantes

Compter de 0 à n-1

Le pattern le plus utilisé en programmation. On part de 0 et on s'arrête **avant** `n`.

```
for (let i = 0; i < 10; i++) {
    console.log(i);
}
// 0 1 2 3 4 5 6 7 8 9  → exactement 10 tours
```

Compter à rebours

```
for (let i = 10; i > 0; i--) {
    console.log(i);
}
// 10 9 8 7 6 5 4 3 2 1
```

Avancer par pas de 2

```
for (let i = 0; i <= 20; i += 2) {
    console.log(i);     // 0 2 4 6 ... 20
}
```

Parcourir un tableau

L'usage le plus courant : passer en revue chaque élément.

```
let fruits = ["pomme", "banane", "cerise"];

for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
```

- `fruits.length` vaut 3 → on tourne pour `i = 0, 1, 2`.
- `fruits[i]` accède au i-ème élément.

Pourquoi `i < fruits.length` et pas `i ≤ 2` ? Parce que le tableau peut grandir. Si on ajoute un fruit, `fruits.length` devient 4 automatiquement, la boucle s'adapte. Avec `i ≤ 2` codé en dur, on oublierait de le mettre à jour. Toujours `< length`.

`for...of` — version simplifiée

Quand on veut juste **chaque élément**, sans s'occuper de l'index, JavaScript moderne propose une syntaxe plus lisible :

```
let fruits = ["pomme", "banane", "cerise"];

for (let fruit of fruits) {
    console.log(fruit);
}
```

Plus court, et **pas de risque d'erreur d'indice**. À utiliser dès que tu n'as pas besoin de `i`.

Quand utiliser laquelle ?

- `for` **classique** : si tu as besoin de l'index (`i`), de sauter des éléments, de parcourir à rebours…
- `for ... of` : si tu veux juste chaque élément un par un.

`break` et `continue`

Deux mots-clés qui modifient le déroulement d'une boucle.

▸ `break` — sortir immédiatement

```
for (let i = 0; i < 100; i++) {
    if (i === 5) {
        break;          // sort de la boucle dès que i vaut 5
    }
    console.log(i);     // 0 1 2 3 4
}
```

Utile quand on cherche quelque chose : dès qu'on l'a trouvé, inutile de continuer.

▸ `continue` — passer au tour suivant

```
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
        continue;       // saute les pairs
    }
    console.log(i);     // 1 3 5 7 9
}
```

`continue` saute le reste du tour courant et passe directement à l'incrément.

Erreurs fréquentes

La boucle infinie

```
// ❌ i ne change jamais → boucle infinie, page bloquée
for (let i = 0; i < 10; ) {
    console.log(i);
}
```

Toujours vérifier que l'incrément fait progresser la condition vers la sortie.

Off-by-one (l'erreur de ±1)

```
// ❌ tourne 11 fois (i = 0, 1, ..., 10), pas 10
for (let i = 0; i <= 10; i++) { ... }

// ✅ tourne 10 fois (i = 0, 1, ..., 9)
for (let i = 0; i < 10; i++) { ... }
```

Règle simple : *"j'ai N éléments, donc* `i < N` *"*.

Modifier le compteur dans la boucle

```
// ❌ Très difficile à lire, à éviter
for (let i = 0; i < 10; i++) {
    if (condition) i += 3;     // saute des tours façon "magie"
    console.log(i);
}
```

Si tu dois sauter des tours, utilise `continue`. Si tu veux changer le rythme, change l'incrément. Modifier `i` au milieu du corps, c'est un piège pour les futurs lecteurs (toi inclus).

Exemple complet : table de multiplication

```
function tableDeMultiplication(n) {
    for (let i = 1; i <= 10; i++) {
        console.log(`${n} × ${i} = ${n * i}`);
    }
}

tableDeMultiplication(7);
```

Et pour faire toutes les tables d'un coup, on imbrique :

```
for (let n = 1; n <= 10; n++) {
    console.log(`--- Table de ${n} ---`);
    for (let i = 1; i <= 10; i++) {
        console.log(`${n} × ${i} = ${n * i}`);
    }
}
```

Une boucle dans une boucle = on multiplie le nombre de tours. 10 × 10 = 100 lignes affichées.

À retenir

- `for (init; condition; increment) { ... }` — trois parties séparées par `;`.
- Pattern standard : `for (let i = 0; i < n; i++)`.
- Pour parcourir un tableau sans avoir besoin de l'index : `for (let element of tableau)`.
- `break` arrête la boucle. `continue` saute au tour suivant.
- Méfiance avec l'off-by-one : `<` pour N tours, `≤` pour N+1 tours.
- Une boucle dont la condition ne devient jamais fausse = boucle infinie, page bloquée.

JavaScript - La boucle while

> *La boucle `while` répète un bloc de code **tant qu'une condition est vraie**. Contrairement au `for`, on ne sait pas forcément à l'avance combien de tours auront lieu — c'est l'outil idéal pour 'continuer jusqu'à ce que quelque chose arrive'.*

La syntaxe

```
while (condition) {
    // instructions répétées tant que la condition est vraie
}
```

C'est plus simple qu'un `for` : **une seule condition**, vérifiée avant chaque tour.

```
let i = 1;

while (i <= 5) {
    console.log(i);
    i++;
}
// 1 2 3 4 5
```

Déroulement :

1. La condition est testée.
2. Si vraie → le bloc s'exécute → retour à l'étape 1.
3. Si fausse → fin de la boucle.

Tu dois modifier la condition dans le corps de la boucle. Si la condition reste vraie indéfiniment, c'est la boucle infinie : page bloquée, onglet à fermer. Toujours s'assurer qu'on **se rapproche** de la sortie.

`for` ou `while` ?

C'est la question classique. La règle simple :

- `for` : *"Je vais répéter N fois"* ou *"je vais parcourir un tableau"*. Le nombre de tours est connu, ou borné par la taille de quelque chose.
- `while` : *"Je continue **jusqu'à ce que** ..."*. Le nombre de tours dépend de ce qui se passe pendant la boucle.

SITUATION	CHOIX RECOMMANDÉ
Afficher les 10 premiers nombres	`for`
Parcourir un tableau	`for ... of`
Demander un mot de passe tant qu'il est incorrect	`while`
Lancer un dé jusqu'à obtenir un 6	`while`
Lire un fichier ligne par ligne jusqu'à la fin	`while`

Techniquement, **tout `for` peut s'écrire avec un `while`** et vice-versa. Le choix est une question de lisibilité.

Exemple typique : ressaisir tant que c'est invalide

```
let motDePasse = prompt("Mot de passe ?");

while (motDePasse !== "azerty123") {
    motDePasse = prompt("Incorrect. Mot de passe ?");
}

alert("Bienvenue !");
```

Combien de tours ? Aucune idée — ça dépend de l'utilisateur. Avec `for`, on devrait fixer une limite arbitraire ; avec `while`, on exprime exactement l'intention.

Exemple : lancer un dé jusqu'à un 6

```
let lancer = 0;
let tentatives = 0;

while (lancer !== 6) {
    lancer = Math.floor(Math.random() * 6) + 1;
    tentatives++;
    console.log(`Lancer ${tentatives} : ${lancer}`);
}

console.log(`Il a fallu ${tentatives} lancers.`);
```

`Math.random()` renvoie un nombre entre 0 et 1. `Math.floor(Math.random() * 6) + 1` donne un entier entre 1 et 6.

La variante `do...while`

Une version qui exécute **le bloc d'abord**, puis vérifie la condition. Garantit **au moins un tour**.

```
let reponse;

do {
    reponse = prompt("Tape 'oui' pour continuer");
} while (reponse !== "oui");
```

Avec `while` classique, si la condition est fausse dès le départ, le bloc n'est jamais exécuté. Avec `do ... while`, on l'exécute au moins une fois. Utile pour les saisies utilisateur : on veut **toujours** poser la question au moins une fois.

Sécurité : la limite anti-boucle-infinie

Quand on prototype, on n'est pas toujours sûr que la condition deviendra fausse. Un garde-fou utile :

```
let lancer = 0;
let tentatives = 0;

while (lancer !== 6 && tentatives < 1000) {
    lancer = Math.floor(Math.random() * 6) + 1;
    tentatives++;
}

if (tentatives >= 1000) {
    console.log("Bug : on a tourné 1000 fois sans obtenir 6 !?");
}
```

La limite (1000 ici) joue le rôle de coupe-circuit. Dès que ça part en vrille, on sort proprement au lieu de bloquer la page.

`break` dans un `while`

Comme dans le `for`, `break` arrête la boucle immédiatement.

```
while (true) {                // boucle a priori infinie
    let saisie = prompt("Mot magique ?");
    if (saisie === "abracadabra") {
        break;                // sortie quand le mot est bon
    }
    console.log("Non, essaie encore.");
}
```

`while (true)` est un raccourci pour *"boucle jusqu'à un* `break` *"*. À utiliser avec parcimonie — c'est puissant mais facile à oublier d'arrêter.

À retenir

- `while (condition) { ... }` — répète **tant que** la condition est vraie.
- Idéal quand on ne sait pas à l'avance combien de tours.
- **Toujours** s'assurer que quelque chose dans le corps fait évoluer la condition, sinon : boucle infinie.
- `do { ... } while (condition)` — exécute au moins une fois avant de tester.
- `break` permet de sortir d'un coup, même depuis un `while (true)`.
- Si tu écris un `while`, demande-toi : *"qu'est-ce qui va arrêter cette boucle ?"*. Si tu ne peux pas répondre, tu as un bug.

JavaScript - Les fonctions

> *Une fonction est un **bloc de code réutilisable** : tu lui donnes un nom, tu lui passes des données, elle exécute ses instructions, et elle peut te renvoyer un résultat. C'est le principal outil pour éviter de copier-coller du code et pour décomposer un programme en morceaux compréhensibles.*

Pourquoi des fonctions ?

Sans fonction, on est vite obligé de **copier-coller**. Imagine que tu veuilles dire bonjour à trois personnes :

```
console.log("Bonjour Alice ! Bienvenue.");
console.log("Bonjour Bob ! Bienvenue.");
console.log("Bonjour Carol ! Bienvenue.");
```

Le jour où tu veux changer "Bonjour" en "Salut", tu modifies trois lignes. Avec dix personnes, dix lignes. C'est fragile et pénible.

Avec une fonction, on écrit **la logique une seule fois** :

```
function direBonjour(nom) {
    console.log(`Bonjour ${nom} ! Bienvenue.`);
}

direBonjour("Alice");
direBonjour("Bob");
direBonjour("Carol");
```

Une modification du message → une seule ligne à changer.

Déclarer une fonction

```
function nomDeLaFonction(parametres) {
    // instructions
}
```

Vocabulaire :

- **Déclarer** une fonction : l'écrire (elle ne fait encore rien).
- **Appeler** une fonction : l'exécuter, en mettant des parenthèses : `nomDeLaFonction()`.
- **Paramètres** : les variables d'entrée que la fonction reçoit.

```
function direBonjour(nom) {     // déclaration
    console.log(`Bonjour ${nom}`);
}

direBonjour("Alice");           // appel
```

Pas de parenthèses = pas d'appel. `direBonjour` (sans `()`) ne fait rien : tu désignes juste la fonction, tu ne l'exécutes pas. C'est utile pour les passer à `addEventListener` par exemple, mais c'est aussi le bug n°1 du débutant : *"pourquoi ma fonction ne s'exécute pas ?"*. Souvent : il manque les `()`.

Paramètres et arguments

- **Paramètres** : les variables déclarées entre les parenthèses de la définition.
- **Arguments** : les valeurs effectivement passées lors de l'appel.

```
function additionner(a, b) {    // a et b sont les paramètres
    console.log(a + b);
}

additionner(3, 5);              // 3 et 5 sont les arguments
```

Une fonction peut avoir **zéro, un, ou plusieurs paramètres** :

```
function saluer() {
    console.log("Hello");
}

function carre(x) {
    console.log(x * x);
}

function moyenne(a, b, c) {
    console.log((a + b + c) / 3);
}
```

`return` — renvoyer un résultat

Une fonction qui se contente d'afficher n'est utile qu'une fois. Pour **réutiliser** un résultat, on utilise `return`.

```
function carre(x) {
    return x * x;
}

let resultat = carre(5);     // resultat vaut 25
console.log(carre(3) + carre(4));   // 9 + 16 = 25
```

`return` :

1. renvoie une valeur à l'endroit où la fonction a été appelée,
2. **arrête immédiatement** la fonction — toute instruction après `return` est ignorée.

```
function estMajeur(age) {
    if (age >= 18) {
        return true;
    }
    return false;             // exécuté seulement si le if est faux
}
```

`console.log` **n'est pas** `return`. `console.log` affiche dans la console mais ne renvoie rien d'exploitable. `return` ne s'affiche pas tout seul mais permet de **réutiliser** la valeur. Ce sont deux outils différents — souvent on commence par `console.log` pour vérifier, puis on remplace par `return` quand la fonction doit servir ailleurs.

Une fonction = un travail

Une bonne fonction fait **une seule chose**, et son nom le décrit clairement.

```
// ✅ Chaque fonction a un rôle clair
function calculerTVA(prix) {
    return prix * 0.21;
}

function formaterPrix(prix) {
    return prix.toFixed(2) + " €";
}

function afficherPrix(prix) {
    const tva = calculerTVA(prix);
    const total = prix + tva;
    console.log(formaterPrix(total));
}
```

Les fonctions s'**enchaînent** naturellement : l'une appelle l'autre. C'est ce qui rend les gros programmes lisibles.

```
// ❌ Une fonction qui fait tout
function faireTout(prix) {
    let tva = prix * 0.21;
    let total = prix + tva;
    let formate = total.toFixed(2) + " €";
    console.log(formate);
    // ... 50 lignes de plus
}
```

Quand le nom commence à devoir énumérer plusieurs verbes (`calculerEtAfficher...`), c'est le signal qu'il faut **découper en plusieurs fonctions**.

Portée des variables

Les variables déclarées **dans** une fonction n'existent **que dans** cette fonction.

```
function calcul() {
    let resultat = 42;
    console.log(resultat);   // OK : 42
}

calcul();
console.log(resultat);       // ❌ ReferenceError : resultat n'existe pas ici
```

C'est une **bonne chose** : chaque fonction a son petit monde, elle ne pollue pas le reste du programme. C'est aussi pour ça qu'on préfère passer des paramètres et utiliser `return` plutôt que des variables globales.

Fonctions fléchées (arrow functions)

Une syntaxe plus courte, très utilisée en JS moderne — surtout avec `addEventListener` et les méthodes de tableau.

```
// Syntaxe classique
function carre(x) {
    return x * x;
}

// Syntaxe fléchée équivalente
const carre = (x) => {
    return x * x;
};

// Encore plus court si le corps est une seule expression
const carre = x => x * x;
```

Trois choses à savoir :

- Les parenthèses autour du paramètre sont **optionnelles** s'il y a un seul paramètre.
- Si le corps est **une seule expression**, on peut omettre les `{}` et le `return` — la valeur est renvoyée automatiquement.
- On utilise très souvent `const` pour stocker une fonction fléchée.

Elles sont particulièrement pratiques en argument d'une autre fonction :

```
bouton.addEventListener("click", () => {
    console.log("Clic !");
});
```

Exemple complet

```
function estMajeur(age) {
    return age >= 18;
}

function tarifEntree(age) {
    if (age < 12) return 5;
    if (age < 18) return 8;
    return 12;
}

function accueillir(nom, age) {
    if (estMajeur(age)) {
        console.log(`Bienvenue ${nom} ! Tarif : ${tarifEntree(age)} €`);
    } else {
        console.log(`Bonjour ${nom}, tarif jeune : ${tarifEntree(age)} €`);
    }
}

accueillir("Alice", 25);     // Bienvenue Alice ! Tarif : 12 €
accueillir("Bob", 14);       // Bonjour Bob, tarif jeune : 8 €
```

Trois fonctions, chacune avec un rôle clair. Pour modifier les tarifs : on touche **uniquement** `tarifEntree`. Pour changer le message d'accueil : **uniquement** `accueillir`.

À retenir

- `function nom(params) { ... }` — déclaration. Pas exécuté tant qu'on ne l'**appelle** pas.
- Appel : `nom(arguments)` — avec les parenthèses.
- `return` renvoie une valeur **et** termine la fonction.
- Une fonction = un travail. Si le nom contient "Et", découpe.
- Les variables locales restent locales — c'est voulu, c'est ce qui rend les fonctions sûres.
- `const nom = () => { ... }` — syntaxe fléchée, surtout utile en argument d'une autre fonction (events, boucles…).

JavaScript - Manipuler le DOM

> *Le DOM (*Document Object Model*) est la représentation de la page HTML que JavaScript voit. Pour rendre une page dynamique, il faut savoir **sélectionner** des éléments, **lire** leur contenu et le **modifier**. C'est la fondation de toute interactivité côté navigateur.*

C'est quoi, le DOM ?

Quand le navigateur charge une page HTML, il en construit une représentation interne : un arbre d'objets, où chaque balise devient un nœud. Cet arbre est le **DOM**.

```
        document
           |
         <html>
         /    \
    <head>   <body>
               |
             <h1>   <p>   <button>
```

JavaScript peut **lire et modifier** cet arbre — et le navigateur met à jour l'affichage en conséquence. C'est ce qui rend les pages dynamiques.

L'objet global qui te donne accès à tout ça : `document`.

Sélectionner un élément

Par identifiant — `getElementById`

L'utilisation la plus simple et la plus rapide.

```
<p id="message">Salut !</p>
```

```
const p = document.getElementById("message");
```

Renvoie l'élément, ou `null` si aucun élément n'a cet `id`. Un `id` doit être **unique** dans la page.

Par sélecteur CSS — `querySelector` / `querySelectorAll`

Plus flexible : on utilise la syntaxe des sélecteurs CSS.

```
// Premier élément qui correspond
document.querySelector("#message");        // par id
document.querySelector(".surbrillance");   // par classe
document.querySelector("p");               // première balise <p>
document.querySelector("ul li.actif");     // sélecteur combiné

// Tous les éléments qui correspondent
document.querySelectorAll(".item");        // une NodeList (≈ tableau)
```

Quand utiliser quoi ?

- `getElementById` — un seul élément précis, par son id. Le plus rapide.
- `querySelector` — un seul élément, sélecteur CSS quelconque.
- `querySelectorAll` — plusieurs éléments. Renvoie une `NodeList` qu'on parcourt comme un tableau.

`querySelector` couvre 90% des cas dès qu'on dépasse les `id`.

Lire et modifier le contenu

Une fois qu'on a un élément, on peut accéder à son contenu :

```
const titre = document.querySelector("h1");

// Lire
console.log(titre.textContent);    // le texte brut
console.log(titre.innerHTML);      // le HTML interne (balises incluses)

// Modifier
titre.textContent = "Nouveau titre";
titre.innerHTML = "Titre <em>en italique</em>";
```

▸ `textContent` ou `innerHTML` ?

PROPRIÉTÉ	EFFET
`textContent`	Texte brut. Les `<` et `>` deviennent du texte affiché. **À privilégier**.
`innerHTML`	Du HTML. Les balises sont interprétées. Puissant **mais risqué** si on injecte du texte venant de l'utilisateur (faille XSS).

Règle simple : `textContent` **par défaut**. N'utilise `innerHTML` que si tu insères du HTML que **tu as écrit toi-même**, jamais du texte tapé par un utilisateur.

Modifier les attributs

Les attributs HTML (`src` , `href` , `alt` , `disabled` …) sont accessibles directement sur l'élément.

```
const img = document.querySelector("#photo");

img.src = "chat.jpg";
img.alt = "Un chat";
img.width = 300;
```

Pour les attributs personnalisés (`data-*`), on passe par `getAttribute` / `setAttribute` :

```
const carte = document.querySelector(".carte");

carte.setAttribute("data-id", 42);
console.log(carte.getAttribute("data-id"));   // "42"
```

Modifier les classes CSS

Plutôt que de toucher au style directement, on **bascule des classes** définies dans le CSS — c'est plus propre et plus maintenable.

```
const bouton = document.querySelector("#valider");

bouton.classList.add("actif");          // ajoute la classe
bouton.classList.remove("actif");       // retire la classe
bouton.classList.toggle("actif");       // ajoute si absente, retire si présente
bouton.classList.contains("actif");     // true / false
```

```
/* dans le CSS */
.actif {
    background-color: green;
    color: white;
}
```

Pour la modification ponctuelle du style en JS (couleur en direct, transition…) voir l'article suivant sur le style.

Créer et insérer des éléments

Pour ajouter dynamiquement des éléments à la page :

```
// 1. Créer
const nouveauP = document.createElement("p");
nouveauP.textContent = "Paragraphe ajouté en JS";
nouveauP.classList.add("note");

// 2. Insérer dans le DOM
document.body.appendChild(nouveauP);     // à la fin du body
```

Variantes :

```
parent.appendChild(enfant);         // ajoute à la fin
parent.prepend(enfant);              // ajoute au début
parent.insertBefore(enfant, autre);  // insère avant un autre élément
enfant.remove();                     // retire du DOM
```

▸ Exemple : remplir une liste

```
<ul id="fruits"></ul>
```

```
const fruits = ["pomme", "banane", "cerise"];
const liste = document.getElementById("fruits");

for (let fruit of fruits) {
    const li = document.createElement("li");
    li.textContent = fruit;
    liste.appendChild(li);
}
```

Trois fruits dans le tableau → trois `<li>` dans la page, sans toucher au HTML.

Lire la valeur d'un champ `<input>`

Pour les formulaires, ce ne sont pas `textContent` ni `innerHTML` qu'on utilise, mais la propriété `.value`.

```
<input id="nom" type="text">
<button id="valider">OK</button>
```

```
const champ = document.getElementById("nom");
const bouton = document.getElementById("valider");

bouton.addEventListener("click", () => {
    const saisie = champ.value;
    console.log(`L'utilisateur a tapé : ${saisie}`);
});
```

`.value` fonctionne pour `<input>`, `<textarea>` et `<select>`. C'est la manière propre de récupérer une saisie utilisateur sur un vrai site (plutôt que `prompt`).

Le piège : exécuter avant le chargement

```
<head>
    <script>
        const titre = document.querySelector("h1");
        titre.textContent = "Nouveau titre";  // ❌ titre est null !
    </script>
</head>
<body>
    <h1>Titre original</h1>
</body>
```

Le script s'exécute **avant** que `<h1>` n'existe dans le DOM → `querySelector` renvoie `null` → erreur en accédant à `.textContent`.

Deux solutions :

1. Mettre le `<script>` **à la fin du** `<body>`, juste avant `</body>`. C'est le réflexe simple.
2. Attendre l'événement `DOMContentLoaded` :

```
document.addEventListener("DOMContentLoaded", () => {
    const titre = document.querySelector("h1");
    titre.textContent = "Nouveau titre";
});
```

À retenir

- Le DOM = arbre d'objets que JavaScript voit, accessible via `document`.
- **Sélection** : `getElementById("id")`, `querySelector("css")`, `querySelectorAll("css")`.
- **Contenu** : `textContent` (sûr, par défaut) vs `innerHTML` (puissant, risqué).

- **Attributs** : `.src` , `.href` , `.value` … ou `getAttribute` / `setAttribute` .
- **Classes** : `classList.add` / `remove` / `toggle` — préférer les classes CSS au style en dur.
- **Création** : `createElement` + `appendChild` pour ajouter des éléments dynamiquement.
- **Champs de formulaire** : `.value` pour lire une saisie.
- **Toujours** placer le `<script>` à la fin du `<body>` (ou utiliser `DOMContentLoaded`).

JavaScript - Manipuler le style et les images avec JavaScript

> *JavaScript peut modifier l'apparence d'une page en direct : changer des styles CSS et swapper des images sans recharger la page.*

Modifier le style d'un élément

Chaque élément HTML possède une propriété `.style` accessible en JS.

```
const titre = document.getElementById("titre");
titre.style.color = "red";
titre.style.fontSize = "2em";
titre.style.display = "none"; // cache l'élément
```

Règle de conversion : les propriétés CSS avec un tiret s'écrivent en camelCase en JS.

CSS	JAVASCRIPT
`font-size`	`fontSize`
`background-color`	`backgroundColor`
`border-radius`	`borderRadius`
`display`	`display`

Les valeurs sont toujours des **strings** :

```
titre.style.fontSize = "24px";   // ✅
titre.style.fontSize = 24;       // ❌ (nombre sans unité, ignoré)
```

▸ Exemple concret

```
<p id="message">Attention !</p>
<button onclick="mettreEnEvidence()">Mettre en évidence</button>

<script>
  function mettreEnEvidence() {
    const msg = document.getElementById("message");
    msg.style.color = "white";
    msg.style.backgroundColor = "crimson";
    msg.style.padding = "8px 16px";
    msg.style.borderRadius = "4px";
  }
</script>
```

Ajouter / retirer une classe CSS

Modifier `.style` directement, c'est bien pour un changement ponctuel. Pour des états réutilisables (actif, erreur, caché…), préfère **manipuler des classes**.

```
/* Dans ton CSS */
.surbrillance {
  background-color: yellow;
  font-weight: bold;
}
```

```
const el = document.getElementById("titre");

el.classList.add("surbrillance");     // ajoute la classe
el.classList.remove("surbrillance");  // retire la classe
el.classList.toggle("surbrillance");  // ajoute si absente, retire si présente
```

> `toggle` *est particulièrement pratique pour les boutons on/off*

(menu, dark mode, accordéon…).

Changer l'image affichée

Une image HTML a un attribut `src`. JS peut le modifier comme n'importe quelle propriété.

```html
<img id="photo" src="chat.jpg" alt="Un chat">
<button onclick="changerImage()">Changer</button>

<script>
  function changerImage() {
    const img = document.getElementById("photo");
    img.src = "chien.jpg";
  }
</script>
```

Exemple : galerie avec un seul `img`

```html
<img id="galerie" src="photo1.jpg" alt="Galerie">
<button onclick="afficher('photo1.jpg')">1</button>
<button onclick="afficher('photo2.jpg')">2</button>
<button onclick="afficher('photo3.jpg')">3</button>

<script>
  function afficher(nomFichier) {
    document.getElementById("galerie").src = nomFichier;
  }
</script>
```

Changer aussi le `alt`

Bonne pratique : mets à jour `alt` en même temps que `src` pour l'accessibilité.

```js
const img = document.getElementById("photo");
img.src = "chien.jpg";
img.alt = "Un chien";
```

À retenir

- `.style.propriété` → modification directe, valeur toujours en string avec unité.
- `.classList.toggle()` → préférable pour des états définis dans le CSS.
- `.src` → change l'image affichée à la volée.

JavaScript - Manipuler le style et les images (exercices 🦾)

> *Voici une série d'exercices pour te permettre de pratiquer.*

Voici 10 consignes centrées sur la manipulation de style HTML via JS — contexte pro, uniquement `document.getElementById`, `element.style.propriété`, conditions `if` simples, `prompt` et variables.

Exercice 01 — Le badge de statut Une startup affiche le statut de son serveur. Récupère l'élément `#status`. Si une variable `serveurEnLigne` vaut `true`, mets son fond en vert et son texte en blanc. Sinon, mets le fond en rouge.

Exercice 02 — Le prix en promotion Un e-commerce applique une remise. Récupère `#prix`. Si une variable `enPromo` vaut `true`, affiche le prix en rouge et en gras. Sinon, affiche-le en noir, style normal.

Exercice 03 — Le niveau de stock Un entrepôt affiche la quantité restante d'un produit. Récupère `#stock`. Si une variable `quantite` est inférieure à 10, mets le texte en orange. Si elle vaut 0, mets le fond en rouge.

Exercice 04 — La carte de membre Une salle de sport affiche les infos d'un membre. Récupère `#carte`. Si une variable `membreVIP` vaut `true`, applique une bordure dorée (`gold`) de 3px et une couleur de fond `#fff8e1`.

Exercice 05 — Le bouton de validation Un formulaire de commande a un bouton `#btnValider`. Si une variable `panierVide` vaut `true`, rends le bouton gris et opaque à 50%. Sinon, mets-le en vert.

Exercice 06 — L'alerte RGPD Un bandeau cookies `#bandeau` doit s'afficher ou non. Demande à l'utilisateur s'il a déjà accepté les cookies (`prompt`). Si la réponse est `"oui"`, cache le bandeau (`display: none`). Sinon, rends-le visible.

Exercice 07 — Le tableau de bord financier Un dashboard affiche un chiffre d'affaires `#ca`. Demande un montant via `prompt`. Si le montant est supérieur à 10 000, affiche-le en vert et en grand (`font-size: 2em`). Sinon, affiche-le en rouge.

Exercice 08 — La date limite Un projet a une deadline. Récupère `#deadline`. Si une variable `joursRestants` est inférieure à 3, mets le texte en rouge et en gras. Si elle est inférieure à 7, mets-le en orange. Sinon, laisse-le en noir.

Exercice 09 — Le profil utilisateur Une app RH affiche un profil `#profil`. Demande le rôle de l'utilisateur via `prompt` (`"admin"` ou autre). Si c'est `"admin"`, applique un fond bleu marine et du texte blanc. Sinon, garde un fond gris clair.

Exercice 10 — Le rapport de performance Un outil analytics affiche un KPI `#kpi`. Récupère via `prompt` un taux de conversion (nombre entre 0 et 100). Si le taux est supérieur à 80, fond vert. Entre 50 et 80, fond orange. En dessous de 50, fond rouge.

JavaScript - Les formulaires

> *Sur un vrai site, on ne demande pas les saisies avec `prompt()` — on utilise des **champs de formulaire** HTML. JavaScript permet de **lire** leur valeur, de **valider** la saisie avant envoi, et d'**intercepter** la soumission pour la traiter sans recharger la page.*

Lire la valeur d'un champ

Tous les champs (`<input>` , `<textarea>` , `<select>`) exposent une propriété `.value` .

```
<input id="nom" type="text">
<button id="valider">OK</button>

<script>
    const champ = document.getElementById("nom");
    const bouton = document.getElementById("valider");

    bouton.addEventListener("click", () => {
        const saisie = champ.value;
        console.log(`Saisie : ${saisie}`);
    });
</script>
```

`champ.value` est **toujours un string**, comme `prompt()` . Pour les champs `<input type="number">` , il faut convertir : `parseInt(champ.value)` .

Cases à cocher et boutons radio

Pour les `checkbox` et les `radio` , on ne lit pas `.value` (qui contient le `value` HTML, fixe) mais `.checked` (true/false).

```
<input type="checkbox" id="newsletter"> Recevoir la newsletter
```

```
const cb = document.getElementById("newsletter");

if (cb.checked) {
    console.log("Inscrit");
} else {
    console.log("Pas inscrit");
}
```

Pour un groupe de boutons radio : on parcourt et on cherche celui qui est coché.

```
<input type="radio" name="taille" value="S"> S
<input type="radio" name="taille" value="M"> M
<input type="radio" name="taille" value="L"> L
```

```
const choisi = document.querySelector('input[name="taille"]:checked');
console.log(choisi.value);   // "S", "M" ou "L"
```

Listes déroulantes

```
<select id="pays">
    <option value="be">Belgique</option>
    <option value="fr">France</option>
    <option value="lu">Luxembourg</option>
</select>
```

```
const select = document.getElementById("pays");
console.log(select.value);   // "be", "fr" ou "lu"
```

Réagir en temps réel : `input`

L'événement `input` se déclenche **à chaque frappe** (ou modification). Idéal pour les compteurs de caractères, les validations en direct, les recherches instantanées.

```
<input id="message" type="text" maxlength="100">
<span id="compteur">100</span>

<script>
    const champ = document.getElementById("message");
    const compteur = document.getElementById("compteur");

    champ.addEventListener("input", () => {
        const restant = 100 - champ.value.length;
        compteur.textContent = restant;
    });
</script>
```

À chaque caractère tapé, le compteur se met à jour. Pas besoin de cliquer sur un bouton.

Intercepter la soumission d'un formulaire

Par défaut, soumettre un formulaire recharge la page (ou redirige). En JavaScript on **intercepte** l'événement `submit` pour traiter la saisie sans rechargement.

```
<form id="contact">
    <input id="email" type="email" required>
    <button type="submit">Envoyer</button>
</form>

<script>
    const form = document.getElementById("contact");

    form.addEventListener("submit", (event) => {
        event.preventDefault();    // bloque l'envoi par défaut

        const email = document.getElementById("email").value;
        console.log(`Email saisi : ${email}`);
        // ici on enverrait à un serveur, on afficherait une confirmation, etc.
    });
</script>
```

`event.preventDefault()` **est la clé.** Sans cet appel, le navigateur soumet quand même le formulaire et recharge la page. Tu vois ton `console.log` *clignoter* puis disparaître. C'est le bug le plus fréquent : *"mon code s'exécute mais ne fait rien"*. Ajoute `preventDefault` et c'est résolu.

Validation côté client

Avant d'envoyer, on vérifie que la saisie est correcte.

```
form.addEventListener("submit", (event) => {
    event.preventDefault();

    const email = document.getElementById("email").value;
    const message = document.getElementById("message").value;

    if (email === "") {
        alert("L'email est obligatoire.");
        return;
    }

    if (!email.includes("@")) {
        alert("Email invalide.");
        return;
    }

    if (message.length < 10) {
        alert("Message trop court (10 caractères minimum).");
        return;
    }

    // Tout est OK
    console.log("Formulaire valide");
});
```

La validation côté client n'est pas une sécurité. N'importe qui peut désactiver JavaScript ou modifier la page. C'est du **confort utilisateur** : feedback rapide, messages clairs. La validation **sérieuse** se fait toujours côté serveur, à la réception des données.

Exemple complet : mini-formulaire d'inscription

```
<form id="inscription">
    <label>Prénom : <input id="prenom" type="text" required></label>
    <label>Âge : <input id="age" type="number" min="0" max="120" required></label>
    <label>
        <input id="cgv" type="checkbox" required> J'accepte les conditions
    </label>
    <button type="submit">S'inscrire</button>
</form>

<div id="resultat"></div>

<script>
    const form = document.getElementById("inscription");
    const resultat = document.getElementById("resultat");

    form.addEventListener("submit", (event) => {
        event.preventDefault();

        const prenom = document.getElementById("prenom").value.trim();
        const age = parseInt(document.getElementById("age").value);
        const cgv = document.getElementById("cgv").checked;

        if (prenom === "") {
            resultat.textContent = "Prénom obligatoire.";
            return;
        }

        if (isNaN(age) || age < 12) {
            resultat.textContent = "Tu dois avoir au moins 12 ans.";
            return;
        }

        if (!cgv) {
            resultat.textContent = "Tu dois accepter les conditions.";
            return;
        }

        resultat.textContent = `Bienvenue ${prenom} !`;
        form.reset();
    });
</script>
```

À noter :

- `trim()` enlève les espaces en début/fin (utile pour les champs texte).

- `parseInt` + `isNaN` pour gérer le cas où l'utilisateur tape n'importe quoi.
- `form.reset()` vide tous les champs après succès.

À retenir

- Lire un champ : `champ.value` (toujours string — convertir si numérique).
- Cases à cocher / radio : `champ.checked` (booléen).
- Soumission : `form.addEventListener("submit", ...)` + `event.preventDefault()`.
- Validation côté client = confort utilisateur. La sécurité, c'est côté serveur.
- `input` event = mise à jour en temps réel à chaque frappe.
- Sur de vrais formulaires, utiliser les attributs HTML (`required` , `min` , `max` , `pattern`) en plus du JS — le navigateur fait déjà 80% du boulot.

JavaScript - Réagir aux évènements

> *Un évènement, c'est tout ce qui se passe sur la page : un clic, une touche pressée, une souris qui survole un bouton. JavaScript permet d'y réagir pour rendre la page interactive.*

Qu'est-ce qu'un évènement ?

Un **évènement** est une action déclenchée sur la page : clic, survol, frappe au clavier, soumission d'un formulaire, chargement de la page…

JavaScript permet d'**écouter** ces évènements et d'exécuter une fonction en réponse.

ÉVÈNEMENT	DÉCLENCHÉ QUAND…
`click`	l'utilisateur clique sur l'élément
`mouseover`	la souris passe au-dessus
`mouseout`	la souris quitte l'élément
`keydown`	une touche du clavier est pressée
`input`	la valeur d'un champ change (à chaque frappe)
`change`	la valeur d'un champ est validée (perte du focus, sélection…)
`submit`	un formulaire est envoyé
`load`	la page (ou une image) a fini de charger

La méthode rapide : `onclick` dans le HTML

Le moyen le plus simple : l'attribut `onclick` directement dans la balise.

```
<button onclick="direBonjour()">Clique ici</button>

<script>
  function direBonjour() {
    alert("Bonjour !");
  }
</script>
```

Ça marche, c'est lisible pour débuter… mais ça **mélange HTML et JavaScript**.

▸ Les limites de `onclick`

```
<button onclick="action1(); action2()">Plusieurs actions</button>
```

- Un seul `onclick` par élément : si tu en écris un deuxième, il **écrase** le premier.
- Le code JavaScript se retrouve coincé dans le HTML, difficile à maintenir.
- Pas pratique pour les évènements autres que `click` (clavier, souris…).

> *Pour un petit prototype, `onclick` dépanne. Pour tout le reste, utilise `addEventListener`.*

La bonne méthode : `addEventListener`

`addEventListener` attache une fonction à un évènement, **depuis le JavaScript**, sans toucher au HTML.

```
<button id="monBouton">Clique ici</button>

<script>
  const bouton = document.getElementById("monBouton");

  bouton.addEventListener("click", function() {
    alert("Bonjour !");
  });
</script>
```

Syntaxe :

```
element.addEventListener("nomDeLEvenement", fonctionAExecuter);
```

- Le nom de l'évènement s'écrit **sans le préfixe** `on` : `"click"`, pas `"onclick"`.
- La fonction peut être anonyme (comme ci-dessus) ou nommée.

▸ Avec une fonction nommée

```
function direBonjour() {
  alert("Bonjour !");
}

bouton.addEventListener("click", direBonjour);
```

⚠️ On passe le **nom** de la fonction, **sans parenthèses**. `direBonjour` (référence) et non `direBonjour()` (appel immédiat).

Pourquoi préférer `addEventListener` ?

1. Plusieurs écouteurs sur le même élément

```
bouton.addEventListener("click", changerCouleur);
bouton.addEventListener("click", afficherMessage);
bouton.addEventListener("click", incrementerCompteur);
```

Les trois fonctions s'exécutent au clic. Impossible avec `onclick`.

2. Séparation HTML / JS (JavaScript "non obstrusif")

Le HTML reste propre, tout le comportement vit dans le `<script>`.

```
<button id="valider">Valider</button>
```

```
document.getElementById("valider")
        .addEventListener("click", validerFormulaire);
```

C'est le principe du **JavaScript non obstrusif** (*unobtrusive JavaScript*) : la structure (HTML), la présentation (CSS) et le comportement (JS) restent dans des fichiers séparés. Le terme est ancien (années 2000) mais l'idée reste d'actualité — c'est la base de tous les frameworks modernes :

- HTML lisible sans JS embarqué partout.
- Comportement modifiable sans toucher au HTML.
- Fichier `.js` externe → mis en cache par le navigateur, réutilisable sur plusieurs pages.

3. Tout type d'évènement

```
champ.addEventListener("input", verifierSaisie);
document.addEventListener("keydown", gererClavier);
window.addEventListener("load", initialiser);
```

4. On peut retirer l'écouteur

```
bouton.removeEventListener("click", direBonjour);
```

(Ne fonctionne qu'avec une fonction **nommée** — pas avec une fonction anonyme.)

L'objet `event`

La fonction appelée reçoit automatiquement un objet `event` contenant les infos sur ce qui s'est passé.

```
bouton.addEventListener("click", function(event) {
  console.log(event);          // tout l'objet
  console.log(event.target);   // l'élément cliqué
  console.log(event.type);     // "click"
});
```

Exemple : connaître la touche pressée

```
document.addEventListener("keydown", function(event) {
  console.log("Touche pressée :", event.key);

  if (event.key === "Enter") {
    alert("Tu as appuyé sur Entrée !");
  }
});
```

Exemple : empêcher le comportement par défaut

```
const formulaire = document.getElementById("monForm");

formulaire.addEventListener("submit", function(event) {
  event.preventDefault(); // bloque l'envoi du formulaire
  console.log("Envoi intercepté !");
});
```

Exemple complet

```
<button id="changeur">Change la couleur</button>
<p id="texte">Regarde-moi changer !</p>

<script>
  const bouton = document.getElementById("changeur");
  const texte = document.getElementById("texte");

  bouton.addEventListener("click", function() {
    texte.style.color = "crimson";
    texte.style.fontWeight = "bold";
  });

  texte.addEventListener("mouseover", function() {
    texte.style.backgroundColor = "yellow";
  });

  texte.addEventListener("mouseout", function() {
    texte.style.backgroundColor = "transparent";
  });
</script>
```

À retenir

- Un **évènement** = une action sur la page (clic, touche, survol…).
- `onclick="..."` dans le HTML : rapide, mais limité et à éviter dès que le projet grandit.
- `element.addEventListener("click", fonction)` : la bonne méthode, plusieurs écouteurs possibles, HTML et JS séparés.
- Le nom de l'évènement s'écrit **sans** `on` : `"click"`, `"keydown"`, `"submit"`…
- La fonction reçoit un objet `event` : `event.target`, `event.key`, `event.preventDefault()`…
- On passe la fonction **sans les parenthèses** : `addEventListener("click", maFonction)`.

JavaScript - Tableaux, objets et JSON

> *Jusqu'ici on a manipulé des valeurs isolées : un nom, un âge, un score. Pour les vrais programmes, on a besoin de structurer des données plus riches : une liste d'élèves, une fiche utilisateur, un panier d'achat. Les **tableaux** stockent des listes, les **objets** stockent des fiches structurées, et **JSON** est le format universel pour les échanger.*

Les tableaux — une liste ordonnée

Un tableau (`Array`) stocke **plusieurs valeurs dans une seule variable**, accessibles par leur position.

```
let fruits = ["pomme", "banane", "cerise"];
```

Chaque case a un **index**, qui commence à **0** (pas à 1 !).

```
console.log(fruits[0]);    // "pomme"
console.log(fruits[1]);    // "banane"
console.log(fruits[2]);    // "cerise"
console.log(fruits.length); // 3
```

Indexation à partir de 0. C'est la source de bug n°1 chez les débutants. Le premier élément est à l'index `0` , le dernier à `length - 1` . Avec 5 éléments → indices 0, 1, 2, 3, 4. Jamais 5.

▸ Modifier un tableau

```
let scores = [10, 20, 30];

scores[1] = 25;            // modifier un élément
scores.push(40);           // ajouter à la fin    → [10, 25, 30, 40]
scores.pop();              // retirer le dernier  → [10, 25, 30]
scores.unshift(5);         // ajouter au début    → [5, 10, 25, 30]
scores.shift();            // retirer le premier  → [10, 25, 30]
```

▸ Parcourir un tableau

```
let fruits = ["pomme", "banane", "cerise"];

// Avec un for classique (utile si on a besoin de l'index)
for (let i = 0; i < fruits.length; i++) {
    console.log(`${i} : ${fruits[i]}`);
}

// Avec for...of (plus lisible si on veut juste les valeurs)
for (let fruit of fruits) {
    console.log(fruit);
}

// Avec forEach (style moderne)
fruits.forEach(fruit => {
    console.log(fruit);
});
```

Les objets — une fiche structurée

Un objet stocke des **paires clé/valeur**. Idéal pour décrire une chose qui a plusieurs caractéristiques.

```
let personne = {
    nom: "Alice",
    age: 16,
    classe: "5TTr",
    estPresent: true
};
```

On accède aux valeurs par leur **nom de clé** (pas par index — on n'a pas d'ordre ici).

```
console.log(personne.nom);        // "Alice"
console.log(personne.age);        // 16
console.log(personne["classe"]);  // "5TTr"  (notation alternative)
```

Dot notation vs bracket notation.

- `personne.nom` — la plus courante, à utiliser quand la clé est connue à l'écriture.
- `personne["nom"]` — utile quand la clé est dans une variable : `personne[champ]`.

▸ Modifier un objet

```
personne.age = 17;              // modifier
personne.email = "a@ex.com";    // ajouter (la clé n'existait pas)
delete personne.estPresent;     // supprimer
```

▸ Tableau d'objets — le pattern le plus utile

C'est la structure de données que tu verras partout : une liste de fiches.

```
let eleves = [
    { nom: "Alice", age: 16, moyenne: 14 },
    { nom: "Bob",   age: 17, moyenne: 11 },
    { nom: "Carol", age: 16, moyenne: 17 }
];

for (let eleve of eleves) {
    console.log(`${eleve.nom} : ${eleve.moyenne}/20`);
}
```

C'est exactement comme ça qu'on représente une base de données simple, une liste de produits, un classement…

Tableau ou objet ?

QUESTION	RÉPONSE
Une liste ordonnée d'éléments du même type ?	**Tableau** (`[]`)
Une fiche avec des champs nommés ?	**Objet** (`{}`)
Une liste de fiches ?	**Tableau d'objets**

```
// ✅ Tableau : liste de scores du même type
let scores = [10, 20, 30, 25, 18];

// ✅ Objet : fiche structurée
let personne = { nom: "Alice", age: 16 };

// ✅ Tableau d'objets : liste de personnes
let eleves = [
    { nom: "Alice", age: 16 },
    { nom: "Bob",   age: 17 }
];

// ❌ Objet avec des clés "0", "1", "2" — utilise un tableau !
let mauvais = { 0: "a", 1: "b", 2: "c" };
```

JSON — le format d'échange

JSON (*JavaScript Object Notation*) est un **format texte** dérivé directement de la syntaxe des objets JavaScript. C'est le format universel pour échanger des données entre :

- une page web et un serveur,
- un programme et un fichier de configuration,
- deux applications différentes,
- même deux langages de programmation différents (Python lit aussi le JSON).

Visuellement, ça ressemble à un objet JS, avec deux différences :

1. Les clés sont **toujours entre guillemets**.
2. Pas de variables, pas de fonctions, pas de commentaires — juste des données.

```
{
    "nom": "Alice",
    "age": 16,
    "classe": "5TTr",
    "matieres": ["math", "info", "anglais"]
}
```

▸ `JSON.stringify` — objet → texte

Pour stocker un objet ou l'envoyer sur le réseau, on doit le convertir en **chaîne de caractères**.

```
let personne = { nom: "Alice", age: 16 };

let texte = JSON.stringify(personne);
console.log(texte);
// '{"nom":"Alice","age":16}'
```

Pratique pour stocker dans `localStorage` ou envoyer via `fetch`.

▸ `JSON.parse` — texte → objet

L'inverse : récupérer un objet utilisable à partir d'un texte JSON (typiquement reçu d'un serveur).

```
let texte = '{"nom":"Alice","age":16}';

let personne = JSON.parse(texte);
console.log(personne.nom);     // "Alice"
console.log(personne.age + 1); // 17
```

`JSON.parse` **plante si le texte n'est pas du JSON valide.** Une virgule en trop, un guillemet manquant, une clé sans guillemets — et tout casse. Si tu reçois du JSON depuis l'extérieur, encadre `JSON.parse` d'un `try / catch` (sujet plus avancé).

Exemple complet : afficher une liste

Combinaison typique : un tableau d'objets, parcouru avec une boucle, qui remplit une page HTML.

```
<ul id="liste"></ul>

<script>
    let eleves = [
        { nom: "Alice", moyenne: 14 },
        { nom: "Bob",   moyenne: 11 },
        { nom: "Carol", moyenne: 17 }
    ];

    const liste = document.getElementById("liste");

    for (let eleve of eleves) {
        const li = document.createElement("li");
        li.textContent = `${eleve.nom} — ${eleve.moyenne}/20`;
        liste.appendChild(li);
    }
</script>
```

Avec dix lignes, on a tout : la structure de données, la boucle, et l'affichage dynamique. C'est le pattern de base de toute application web.

À retenir

- **Tableau** `[ ]` : liste ordonnée, accès par index commençant à **0**, `length` donne la taille.
- **Objet** `{ }` : fiche structurée, accès par nom de clé : `obj.cle`.
- **Tableau d'objets** : combinaison ultra-courante (liste d'éléments, panier, classement…).
- **JSON** : format texte universel, clés entre guillemets obligatoires.
- `JSON.stringify(obj)` → texte. `JSON.parse(texte)` → objet.
- Liste de choses similaires → tableau. Une chose avec plusieurs champs → objet. Liste de fiches → tableau d'objets.

JavaScript - Projet intégrateur

> *Tu as vu toutes les briques : variables, conditions, boucles, fonctions, DOM, événements, données structurées. Le projet final, c'est mettre tout ensemble pour produire **une vraie page web interactive** sur le sujet de ton choix.*

Objectif

Concevoir et réaliser une **page web interactive de A à Z** (HTML + CSS + JS) sur un sujet de ton choix, puis la **présenter** en expliquant tes choix techniques.

Le projet doit mobiliser **au minimum** les notions suivantes :

- Au moins une **saisie utilisateur** (champ `<input>` ou `prompt`).
- Au moins une **condition** (`if / else`).
- Au moins une **boucle** (`for` ou `while`).
- Au moins une **fonction** déclarée par toi (en plus de celles appelées par les events).
- Au moins une **manipulation du DOM** (changer du texte, une couleur, ajouter un élément).
- Au moins un **événement** (`click`, `input`, `submit`…).
- Une structure de données : **tableau** ou **objet** (idéalement un tableau d'objets).

Suggestions de projets

Quelques idées si tu manques d'inspiration. **Tu n'es pas obligé** d'en choisir une — un projet personnel motivant vaut mieux qu'un projet imposé.

Niveau accessible

- **Quiz à choix multiple** — questions chargées depuis un tableau d'objets, score affiché à la fin.
- **Calculatrice** — boutons, opérateur, deux opérandes, affichage du résultat.
- **Convertisseur d'unités** — euros↔dollars, km↔miles, °C↔°F.
- **Liste de tâches (to-do)** — ajouter, cocher, supprimer des tâches.
- **Devine le nombre** — l'ordinateur tire un nombre, l'utilisateur essaie de le trouver avec indices.

Niveau ambitieux

- **Pierre/feuille/ciseaux** contre l'ordinateur, avec score sur plusieurs manches.
- **Memory** (jeu d'appariement de cartes) avec timer.
- **Mini-galerie photo** avec filtrage par catégorie.
- **Carnet de notes** persistant dans `localStorage`.
- **Mini-Pokédex** : liste de créatures, recherche, fiche détaillée.

Étapes recommandées

1. Cahier des charges (1 page maximum)

Avant d'écrire une ligne de code :

- **Quoi ?** Une phrase qui décrit ce que fait l'application.
- **Pour qui ?** Une cible imaginaire (joueur, élève, prof, client…).
- **Fonctions essentielles** (3 à 5 max, sans détails techniques).
- **Maquette papier** — un schéma rapide de l'écran. Pas besoin d'être beau, juste lisible.

2. Squelette HTML + CSS statique

Une page qui **ressemble** au produit fini, **sans aucune logique JS**. Tu mets juste les éléments en place : titres, champs, boutons, zones d'affichage. Tu vérifies dans le navigateur que ça se présente correctement.

3. JavaScript par petits incréments

Une fonctionnalité à la fois. Toujours dans cet ordre :

1. Écris la fonction.
2. **Teste-la dans la console** avec un cas simple.
3. Branche-la sur l'événement (clic, soumission…).
4. **Vérifie dans le navigateur** que ça marche.
5. Passe à la suivante.

Ne code pas 3 fonctionnalités d'affilée sans tester. C'est le piège classique : tu écris 100 lignes, rien ne marche, et tu ne sais plus quelle ligne casse. Un truc à la fois, on teste, on continue. Si tu bloques 15 minutes : `console.log` partout.

4. Polissage

Une fois que tout marche :

- Vérifie les **cas limites** : champ vide, valeur négative, click avant saisie…
- **Renomme** tes variables et fonctions si certaines sont devenues vagues.
- **Retire les** `console.log` inutiles.
- Ajoute un peu de CSS pour que ce soit présentable.

5. Préparation de la présentation

- **5 minutes maximum** de démonstration.
- Prévois un **scénario d'utilisation** : ce que tu vas montrer, dans quel ordre.
- Anticipe les **questions techniques** : *"pourquoi un objet et pas un tableau ici ?"*, *"que se passe-t-il si on clique sans rien taper ?"*.
- Sois prêt à **lire un morceau de ton code** et l'expliquer.

Grille d'évaluation

CRITÈRE	PONDÉRATION
Le projet fonctionne sans erreur visible	/20
Toutes les notions requises sont présentes (saisie, condition, boucle, fonction, DOM, event, structure de données)	/20
Le code est **lisible** : noms clairs, indentation, pas de copié-collé évident	/20
Les cas limites sont gérés (saisie vide, valeur invalide…)	/10
Présentation orale claire, capacité à expliquer ses choix techniques	/20
Originalité / soin / ambition raisonnable	/10

Le critère le plus important est la lisibilité. Un projet petit et propre vaut mieux qu'un projet ambitieux et illisible. Si tu dois copier-coller le même code à trois endroits, c'est qu'il manque une fonction. Si une variable s'appelle `x`, c'est qu'il manque un nom. Code pour le toi de la semaine prochaine.

Quelques règles d'autonomie

Tu vas inévitablement rencontrer un problème que le cours ne couvre pas. Bonne nouvelle : c'est exactement ce que fait tout développeur, tous les jours.

- **Documentation MDN** (`developer.mozilla.org`) — la référence officielle. En français pour la plupart des pages. Toujours commencer par là.
- **StackOverflow** — une question = souvent déjà posée. Lis les **questions** autant que les réponses.
- **Console du navigateur** — quand un truc ne marche pas, l'erreur rouge te dit où.
- **Le débogueur** (onglet Sources) — pour mettre des points d'arrêt et exécuter pas à pas.

Une bonne recherche c'est : `javascript [problème exact]` + souvent l'année (`2024`) pour éviter les vieux articles avec `var` et `==`.

À retenir

- Un projet, c'est : un cahier des charges → HTML/CSS statique → JS incrémental → polissage → présentation.
- **Une fonctionnalité à la fois**, testée avant de passer à la suivante.
- Lisibilité > ambition. Mieux vaut un quiz qui marche qu'un MMO qui plante.
- Quand tu bloques : `console.log` partout, MDN, et reformuler le problème par écrit.
- À la présentation, sois prêt à expliquer **pourquoi** tu as fait tel choix, pas juste **ce que** ton code fait.

Bonne chance — et amuse-toi.

Création de pages Web avec Python

introduction

Flask - Les routes

> *Dans le développement web, l'une des premières choses que l'on doit maîtriser est la manière dont les requêtes sont gérées par le serveur, et comment elles sont redirigées vers le bon traitement. Lorsque l'on travaille avec Flask, un micro-framework pour Python, cet aspect est géré par un concept central : les **routes**. Cet article vous expliquera en détail ce que sont les **routes**, comment elles fonctionnent, et comment les utiliser efficacement dans vos applications web.*

Qu'est-ce qu'une "Route" dans Flask ?

En termes simples, une **route** est une association entre une **URL** et une **fonction** Python dans votre application Flask. Lorsque Flask reçoit une requête HTTP, il regarde l'URL et essaie de faire correspondre cette URL à une route définie dans votre code. Si une correspondance est trouvée, Flask exécute la fonction associée et envoie la réponse générée au navigateur de l'utilisateur.

Exemple basique de route

Voici un exemple simple pour mieux comprendre ce concept :

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Bienvenue sur la page d'accueil !"

if __name__ == '__main__':
    app.run(debug=True)
```

Dans cet exemple :

- `@app.route('/')` : C'est un décorateur qui lie l'URL racine (`/`) à la fonction `home()` .
- `home()` : Cette fonction est exécutée lorsque l'utilisateur visite l'URL racine du site web (par exemple, `http://localhost:5000/`). Elle renvoie une chaîne de caractères qui sera affichée dans le navigateur.

Chaque fois que Flask reçoit une requête pour l'URL `/` , il sait qu'il doit appeler la fonction `home()` et renvoyer la réponse correspondante.

Comment définir des routes en Flask ?

Utilisation des Décorateurs

Les routes en Flask sont principalement définies avec un **décorateur** : `@app.route()`. Ce décorateur est placé juste au-dessus d'une fonction, indiquant à Flask que cette fonction doit être exécutée lorsqu'un utilisateur accède à une URL spécifique.

Gérer plusieurs méthodes HTTP

Par défaut, une route en Flask ne répond qu'aux requêtes HTTP de type **GET**. Cela signifie que si un utilisateur entre une URL dans son navigateur ou clique sur un lien, Flask déclenchera la fonction liée à cette route avec une requête `GET`. Cependant, il est possible de spécifier d'autres méthodes HTTP, comme **POST**, **PUT**, ou **DELETE**, en fonction des besoins de votre application.

Voici un exemple où une route accepte à la fois les méthodes `GET` et `POST` :

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return "Tentative de connexion"
    return "Page de connexion"
```

Dans cet exemple :

- Si la méthode de la requête est **GET**, Flask renverra simplement "Page de connexion".
- Si la méthode est **POST**, par exemple lorsqu'un formulaire de connexion est soumis, la réponse sera "Tentative de connexion".

Les raccourcis `@app.get` et `@app.post`

Flask offre des **raccourcis** pour définir rapidement des routes qui utilisent spécifiquement les méthodes HTTP **GET** et **POST**. Plutôt que d'utiliser le décorateur général `@app.route()` et de préciser les méthodes acceptées via l'argument `methods`, vous pouvez utiliser les décorateurs `@app.get()` et `@app.post()`.

`@APP.GET()`

Ce raccourci est équivalent à `@app.route( ... , methods=['GET'])`. Il est utilisé pour les routes qui acceptent uniquement les requêtes **GET**, typiquement pour afficher des pages ou des informations. Par exemple :

```
@app.get('/')
def home():
    return "Page d'accueil"
```

Dans cet exemple, la route définie avec `@app.get('/')` accepte uniquement les requêtes GET, ce qui est idéal pour des pages statiques ou des ressources à lire.

`@APP.POST()`

De la même manière, `@app.post()` est un raccourci pour `@app.route( ... , methods=['POST'])`. Ce décorateur est utilisé lorsque la route doit accepter uniquement des requêtes **POST**, souvent utilisées pour soumettre des formulaires ou envoyer des données au serveur. Voici un exemple :

```
@app.post('/login')
def login():
    return "Traitement du formulaire de connexion"
```

Avec ce décorateur, la fonction `login()` ne sera accessible que pour les requêtes POST, ce qui est une bonne pratique pour les actions sensibles comme la soumission de formulaires.

AVANTAGES DES RACCOURCIS

L'utilisation de ces raccourcis permet de rendre votre code plus **lisible** et plus **explicite** en ce qui concerne le type de requête HTTP que la route gère. Cela simplifie également la structure de votre code en éliminant la nécessité de spécifier l'argument `methods` pour les requêtes courantes GET et POST, ce qui est particulièrement utile dans des projets de petite à moyenne envergure.

▸ Gestion des erreurs 404 avec des routes

En développement web, il est courant que les utilisateurs accèdent à des pages inexistantes, ce qui génère une erreur **404**. Flask vous permet de gérer cela facilement en définissant une route spéciale pour cette erreur :

```
@app.errorhandler(404)
def page_not_found(error):
    return "Page non trouvée", 404
```

Ici, la fonction `page_not_found()` sera appelée chaque fois qu'une route non définie est accédée, et Flask renverra une réponse avec un message personnalisé et le code d'erreur HTTP 404.

Exemples complets d'application Flask avec des routes

Maintenant que nous avons vu plusieurs aspects des routes en Flask, voici un exemple d'application complète avec plusieurs routes :

```python
from flask import Flask, request

app = Flask(__name__)

# Route pour la page d'accueil
@app.route('/')
def home():
    return "Page d'accueil"

# Route dynamique pour afficher un profil utilisateur
@app.route('/user/admin')
def user_profile():
    return f"Profil de 'Admin'"

# Route pour gérer les connexions avec GET et POST
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return "Connexion réussie"
    else:
        return "Formulaire de connexion"

# Route pour les erreurs 404
@app.errorhandler(404)
def page_not_found(error):
    return "Cette page n'existe pas", 404

if __name__ == '__main__':
    app.run(debug=True)
```

Ce que fait cette application :

- **Route racine (`/`)** : Renvoie "Page d'accueil".
- **Route utilisateur (`/user/admin`)** : Affiche le profil de l'utilisateur `admin`.
- **Route de connexion (`/login`)** : Affiche un formulaire de connexion pour une requête GET et traite la connexion pour une requête POST.
- **Gestion des erreurs 404** : Si un utilisateur tente d'accéder à une page non définie, il recevra un message "Cette page n'existe pas" avec un statut 404.

Conclusion

Les **routes** sont l'un des concepts les plus fondamentaux dans Flask et permettent de **définir comment votre application web doit répondre aux différentes requêtes HTTP**. En comprenant comment lier des URLs à des fonctions Python, gérer des paramètres dynamiques, et supporter différentes méthodes HTTP, vous serez capable de construire des applications web plus complexes et interactives.

Il est également important de faire attention à l'**ordre de définition des routes** dans votre application pour éviter des conflits entre des routes similaires, en définissant toujours les routes les plus spécifiques avant les routes plus générales.

Flask offre une grande flexibilité tout en restant léger et simple à utiliser. En maîtrisant l'utilisation des routes, vous poserez les bases nécessaires pour concevoir des applications web robustes et performantes. Bonne chance dans vos projets Flask !

Flask - Les routes | Exercices

> *Voici quatre exercices progressifs et complets pour apprendre à utiliser les routes dans Flask. Chaque exercice introduit un nouveau concept, et vous aidera à comprendre comment gérer les routes et les requêtes HTTP dans une application web Flask.*

Exercice 1 : Création de routes de base

Objectif

Dans cet exercice, vous allez créer une application Flask avec plusieurs routes simples qui affichent des messages différents selon l'URL visitée.

Instructions

1. Créez un projet Flask.
2. Ajoutez trois routes distinctes dans l'application Flask :
   - Une route pour la page d'accueil (`/`) qui affiche le message "Bienvenue sur la page d'accueil".
   - Une route pour une page "contact" (`/contact`) qui affiche le message "Page de contact".
   - Une route pour une page "à propos" (`/about`) qui affiche le message "À propos de nous".

Code de base

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Bienvenue sur la page d'accueil"

@app.route('/contact')
def contact():
    return "Page de contact"

@app.route('/about')
def about():
    return "À propos de nous"

if __name__ == '__main__':
    app.run(debug=True)
```

Objectifs d'apprentissage

- Comprendre la création de routes avec le décorateur `@app.route()`.
- Gérer plusieurs routes dans une application Flask.
- Exécuter et tester l'application dans un navigateur.

Exercice 2 : Routes avec paramètres dynamiques

Objectif

Dans cet exercice, vous allez ajouter une route dynamique qui accepte un paramètre dans l'URL et renvoie une réponse personnalisée.

Instructions

1. Ajoutez une nouvelle route qui accepte un paramètre `nom` dans l'URL. Par exemple, si l'utilisateur visite `/bonjour/John`, la page affichera "Bonjour, John".
2. Faites en sorte que cette route fonctionne avec n'importe quel nom.

Code de base

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Bienvenue sur la page d'accueil"

@app.route('/bonjour/<nom>')
def bonjour(nom):
    return f"Bonjour, {nom}"

if __name__ == '__main__':
    app.run(debug=True)
```

Test

- Visitez l'URL `/bonjour/John` et vérifiez que le message est bien personnalisé.
- Essayez d'autres noms comme `/bonjour/Marie` ou `/bonjour/Pierre`.

Objectifs d'apprentissage

- Utiliser des paramètres dynamiques dans les routes.
- Générer des réponses personnalisées en fonction de l'URL visitée.

Exercice 3 : Routes avec des types de paramètres

Objectif

Dans cet exercice, vous allez créer une route qui accepte des paramètres de types différents (entiers et chaînes de caractères) et gérez le type des données dans l'URL.

Instructions

1. Créez une nouvelle route qui accepte un **ID** utilisateur (un entier) et un nom d'utilisateur (une chaîne de caractères).
2. Affichez un message du type "Profil de l'utilisateur avec ID: `<id>` et nom: `<nom>`".
3. Utilisez le convertisseur `int` pour forcer l'ID à être un entier.

Code de base

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Bienvenue sur la page d'accueil"

@app.route('/profil/<int:id>/<nom>')
def user_profile(id, nom):
    return f"Profil de l'utilisateur avec ID: {id} et nom: {nom}"

if __name__ == '__main__':
    app.run(debug=True)
```

Test

- Visitez `/profil/1/John` pour tester la route.
- Essayez `/profil/2/Marie` pour voir le comportement avec un autre nom et ID.

Objectifs d'apprentissage

- Comprendre l'utilisation des types de paramètres dans les routes.
- Gérer des types de données différents dans une même URL (entiers et chaînes de caractères).

Exercice 4 : Routes avec méthodes HTTP (GET et POST)

Objectif

Dans cet exercice, vous allez créer une route qui gère les méthodes **GET** et **POST**. Vous ajouterez un formulaire HTML pour soumettre des données et les traiter avec Flask.

Instructions

1. Créez une route `/login` qui accepte les méthodes `GET` et `POST`.
2. Lorsque la méthode est `GET`, affichez un formulaire de connexion simple avec un champ pour le nom d'utilisateur.
3. Lorsque la méthode est `POST`, récupérez le nom d'utilisateur soumis et affichez "Connexion réussie pour `<nom d'utilisateur>`".

Code de base

```
from flask import Flask, request, render_template_string

app = Flask(__name__)

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        return f"Connexion réussie pour {username}"
    return '''
        <form method="POST">
            Nom d'utilisateur: <input type="text" name="username">
            <input type="submit" value="Se connecter">
        </form>
    '''

if __name__ == '__main__':
    app.run(debug=True)
```

Test

- Accédez à la page `/login` pour voir le formulaire.
- Soumettez un nom d'utilisateur et vérifiez que la connexion est réussie.
- Testez avec plusieurs noms d'utilisateur.

Objectifs d'apprentissage

- Gérer plusieurs méthodes HTTP (`GET` et `POST`) avec une même route.
- Récupérer des données soumises via un formulaire HTML.
- Utiliser la méthode `POST` pour envoyer des données depuis le navigateur vers le serveur.

Ces exercices couvrent progressivement les principaux concepts liés aux routes dans Flask : la création de routes basiques, l'utilisation de paramètres dynamiques, la gestion de types de données dans les URL, et la gestion des différentes méthodes HTTP. Ils sont conçus pour guider les débutants dans l'apprentissage des fonctionnalités essentielles de Flask.

Flask - Les routes avec variables

> *Une des fonctionnalités puissantes de Flask est la capacité à gérer des **variables dynamiques** dans les routes. Cela signifie que vous pouvez **inclure des parties variables** dans l'URL et les transmettre à la fonction associée pour qu'elle les traite.*

Une des fonctionnalités puissantes de Flask est la capacité à gérer des **variables dynamiques** dans les routes. Cela signifie que vous pouvez **inclure des parties variables** dans l'URL et les transmettre à la fonction associée pour qu'elle les traite.

Par exemple, imaginons que vous souhaitiez afficher le profil d'un utilisateur selon son nom dans l'URL :

```
@app.route('/user/<username>')
def show_user_profile(username):
    return f"Profil de l'utilisateur : {username}"
```

- `<username>` dans l'URL représente une varianble dynamique.
- Cette variable est ensuite passée à la fonction `show_user_profile(username)` qui l'utilise pour générer une réponse personnalisée.

Ainsi, si un utilisateur accède à l'URL `/user/John`, Flask exécutera `show_user_profile('John')`, et la réponse sera : "Profil de l'utilisateur : John".

L'importance de l'ordre de définition des routes

Dans Flask, **l'ordre de définition des routes est crucial** pour assurer le bon fonctionnement de l'application. Flask traite les routes dans l'ordre où elles sont définies dans votre code. Lorsqu'une requête HTTP est reçue, Flask compare l'URL demandée avec chaque route définie, dans l'ordre, jusqu'à ce qu'il trouve une correspondance.

Cela signifie que si vous définissez une route plus générale avant une route plus spécifique, la route générale sera exécutée en premier, même si une route plus précise existe. Cela peut causer des problèmes si vous attendez un comportement spécifique pour certaines URL.

Exemple d'ordre incorrect

Prenons un exemple pour illustrer cela :

```python
@app.route('/user/<username>')
def user_profile(username):
    return f"Profil de {username}"

@app.route('/user/admin')
def admin_profile():
    return "Profil de l'administrateur"
```

Dans cet exemple, Flask ne pourra jamais accéder à la route `/user/admin` même si elle est définie. Pourquoi ? Parce que la route `/user/<username>` capture déjà toutes les URL qui commencent par `/user/`, y compris `/user/admin`. La route dynamique est donc prioritaire et interceptée avant que Flask n'atteigne la route spécifique `/user/admin`.

▸ Comment éviter ce problème

Pour éviter cela, il est important de définir d'abord les routes les plus **spécifiques** avant les routes plus **générales**. Voici comment corriger l'exemple précédent :

```python
@app.route('/user/admin')
def admin_profile():
    return "Profil de l'administrateur"

@app.route('/user/<username>')
def user_profile(username):
    return f"Profil de {username}"
```

Dans ce cas, Flask vérifie d'abord si l'URL correspond à `/user/admin`. Si ce n'est pas le cas, il passe à la route plus générale `/user/<username>`, qui capturera toutes les autres URLs commençant par `/user/`.

Conversion des variables dans les chemins

Lorsque vous définissez des routes dynamiques en Flask, vous pouvez inclure des **variables** dans les chemins, mais aussi spécifier le **type de données** attendu pour ces variables. Par défaut, Flask considère tous les variables dynamiques comme des chaînes de caractères, mais vous pouvez utiliser des convertisseurs de types pour forcer des types spécifiques comme des entiers ou des flottants. Ces convertisseurs permettent de contrôler et valider les données directement au niveau de l'URL.

▸ Exemple de conversion

Voici comment utiliser un convertisseur de type pour s'assurer qu'une variable est un entier :

```python
@app.route('/user/<int:user_id>')
def show_user_profile(user_id):
    return f"Profil de l'utilisateur avec ID : {user_id}"
```

Dans cet exemple, le chemin `/user/<int:user_id>` force Flask à accepter uniquement des **entiers** pour la variable `user_id`. Si l'utilisateur tente d'accéder à `/user/abc`, Flask renverra automatiquement une erreur 404, car la variable `abc` n'est pas un entier.

Convertisseurs disponibles

Flask propose plusieurs convertisseurs prédéfinis :

- `<string>` (par défaut) : Correspond à une chaîne de caractères.
- `<int>` : Accepte uniquement des entiers.
- `<float>` : Accepte uniquement des nombres à virgule flottante.
- `<path>` : Accepte des chaînes de caractères avec des barres obliques, utile pour capturer des sous-chemins.

Utilité des conversions

Ces conversions facilitent la gestion des données directement dans l'URL, réduisant le besoin de validation manuelle dans votre code, ce qui améliore la robustesse et la sécurité de l'application. Elles permettent également d'améliorer l'expérience utilisateur en évitant des erreurs de type lors des requêtes.

Flask - Les fichiers statiques

> *Dans le développement web, il est essentiel de gérer correctement les fichiers **statiques** pour offrir une expérience utilisateur fluide et interactive. Les fichiers statiques regroupent tout ce qui ne change pas d'une requête à l'autre, comme les images, les fichiers CSS, JavaScript, ou les polices de caractères. Flask, bien que léger et minimaliste, offre une gestion intégrée des fichiers statiques, simplifiant ainsi leur utilisation. Cet article explore en détail l'utilisation des fichiers statiques avec Flask, leur rôle pendant le développement, et pourquoi leur gestion est cruciale pour construire des applications web performantes et esthétiques.*

Qu'est-ce qu'un fichier "statique" ?

Un fichier **statique** est un fichier qui **n'est pas généré ou modifié dynamiquement par le serveur**. Contrairement aux pages ou aux données générées par des templates ou des bases de données, les fichiers statiques restent constants. Quelques exemples typiques :

- **Images** : JPEG, PNG, GIF.
- **Feuilles de style CSS** : Utilisées pour définir la présentation graphique du site.
- **Scripts JavaScript** : Code côté client qui ajoute des interactions dynamiques aux pages.
- **Polices web** : Pour améliorer la typographie.

Ces fichiers sont chargés directement par le navigateur et ne nécessitent pas de traitement particulier côté serveur, hormis l'envoi du fichier au client. Les fichiers statiques sont pré-enregistrés et restent toujours les mêmes.

Pourquoi les fichiers statiques sont-ils importants ?

Les fichiers statiques jouent un rôle majeur dans l'aspect visuel et interactif d'une application web. Par exemple, sans les fichiers CSS, une page web serait une simple collection de texte brut sans style, et sans JavaScript, il serait impossible d'ajouter des interactions complexes comme la validation de formulaires ou l'animation de menus.

Pendant le **développement**, la gestion des fichiers statiques est cruciale pour :

1. **Organiser le projet** : Centraliser et bien organiser vos ressources statiques vous permet de maintenir un code propre et évolutif.
2. **Accélérer le chargement des pages** : Bien gérer les fichiers statiques permet de réduire la taille des ressources envoyées au navigateur, améliorant ainsi la vitesse de chargement.
3. **Améliorer l'expérience utilisateur** : Les styles et les scripts permettent de créer une interface conviviale, agréable et interactive.

Gestion des fichiers statiques avec Flask

Flask simplifie la gestion des fichiers statiques via un dossier nommé `static/` à la racine de votre projet. Par défaut, Flask sait où trouver les fichiers statiques et les rend directement accessibles depuis l'URL `/static/<nom_du_fichier>`.

Exemple de structure de projet

Voici comment structurer un projet Flask typique avec des fichiers statiques :

```
mon_projet/
│
├── static/
│   ├── css/
│   │   └── styles.css
│   ├── js/
│   │   └── script.js
│   └── images/
│       └── logo.png
│
└── app.py
```

- `static/` : Ce dossier contient tous les fichiers statiques de l'application. À l'intérieur, vous pouvez organiser les sous-dossiers en fonction du type de fichiers (CSS, JavaScript, images, etc.).
- `app.py` : Fichier principal contenant le code Flask.

Comment utiliser les fichiers statiques

Lorsque vous placez des fichiers dans le dossier `static/`, ils deviennent accessibles via une URL simple. Par exemple :

- Un fichier CSS placé dans `static/css/styles.css` sera accessible à l'URL `/static/css/styles.css`.
- Une image dans `static/images/logo.png` sera accessible à l'URL `/static/images/logo.png`.

Vous pouvez directement référencer ces fichiers dans vos pages HTML.

Exemple d'inclusion d'un fichier CSS dans une page HTML

Imaginons que vous ayez un fichier **CSS** pour le style de votre site. Voici comment l'inclure dans votre fichier HTML :

```html
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ma Page</title>
    <!-- Inclusion du fichier CSS -->
    <link rel="stylesheet" href="/static/css/styles.css">
</head>
<body>
    <h1>Bienvenue sur mon site web</h1>
</body>
</html>
```

Dans cet exemple, le fichier `styles.css` est référencé à l'URL `/static/css/styles.css`. Flask servira automatiquement ce fichier CSS lorsqu'un utilisateur visitera votre site.

Inclusion d'images dans une page HTML

Si vous avez un fichier image, par exemple un logo, dans le dossier `static/images/`, vous pouvez l'inclure ainsi dans votre HTML :

```html
<img src="/static/images/logo.png" alt="Logo du site">
```

Le chemin de l'image est `/static/images/logo.png`, et Flask servira ce fichier lorsque la page sera chargée.

Utilisation de JavaScript

Les fichiers **JavaScript** sont utilisés pour ajouter des fonctionnalités interactives à votre site web. Par exemple, voici comment inclure un fichier JavaScript :

```html
<script src="/static/js/script.js"></script>
```

Ce fichier `script.js` sera exécuté chaque fois que la page se charge.

Exemple complet

Voici un exemple de page HTML qui utilise des fichiers statiques pour le style, les images, et les fonctionnalités JavaScript :

```html
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Page complète avec fichiers statiques</title>
    <!-- CSS -->
    <link rel="stylesheet" href="/static/css/styles.css">
</head>
<body>
    <h1>Bienvenue sur mon site</h1>

    <!-- Image -->
    <img src="/static/images/logo.png" alt="Logo">

    <!-- JavaScript -->
    <button id="monBouton">Cliquez ici</button>
    <script src="/static/js/script.js"></script>
</body>
</html>
```

Dans cet exemple :

- Le fichier **CSS** stylise la page.
- Une **image** est affichée sous forme de logo.
- Un **script JavaScript** est inclus pour ajouter une fonctionnalité dynamique au bouton.

Les bonnes pratiques pour l'utilisation des fichiers statiques

Lorsque vous travaillez avec des fichiers statiques dans Flask, voici quelques bonnes pratiques à suivre :

1. **Organisez bien vos fichiers** : Placez vos fichiers CSS, JavaScript, et images dans des sous-dossiers pour garder votre projet bien structuré.
2. **Nommez vos fichiers clairement** : Utilisez des noms de fichiers explicites pour faciliter leur gestion. Par exemple, utilisez `styles.css` pour un fichier de style, et non `main.css`, qui pourrait être trop vague.
3. **Minifiez vos fichiers avant la mise en production** : Réduisez la taille de vos fichiers CSS et JavaScript avant de déployer votre application en production, pour améliorer les temps de chargement.
4. **Évitez les fichiers inutiles** : Supprimez les fichiers statiques qui ne sont plus utilisés afin de ne pas alourdir votre projet inutilement.

Conclusion

Les fichiers statiques dans le développement web permettent d'améliorer l'apparence et l'interactivité de votre site. Avec Flask, il est très facile de gérer ces fichiers via le dossier `static/`, où vous pouvez organiser vos fichiers CSS, JavaScript et images. En les référant directement via des chemins simples dans vos pages HTML, vous pouvez rapi-

dement et facilement créer des interfaces utilisateur riches et interactives.

N'oubliez pas que bien organiser vos fichiers statiques et suivre les bonnes pratiques vous aidera à maintenir un projet propre et performant, surtout à mesure que votre application grandit.

Flask - url_for

> *Tu vas apprendre à utiliser `url_for` pour **générer automatiquement les URL de ton application** en fonction du nom de la fonction de route.*

Notions essentielles

Qu'est-ce que `url_for` ?

- C'est une fonction qui retourne l'URL associée à une **fonction de route**.
- Tu ne dois pas écrire les chemins en dur : Flask les construit pour toi.

Exemple simple

```
from flask import Flask, url_for

app = Flask(__name__)

@app.route('/')
def home():
    return "Bienvenue !"

with app.test_request_context():
    print(url_for("home"))  # Affiche "/"
```

Avec des paramètres

Si ta route a un paramètre, tu peux le passer à `url_for` :

```
@app.route('/user/<name>')
def user(name):
    return f"Bonjour {name}"

with app.test_request_context():
    print(url_for("user", name="Alice"))
    # Résultat : "/user/Alice"
```

Pourquoi l'utiliser ?

- Tu évites les erreurs si l'URL change.
- Ton code est plus clair et plus facile à maintenir.

- Tu peux générer des liens dynamiques dans tes templates Jinja2 :

```
<a href="{{ url_for('home') }}">Accueil</a>
```

À retenir

- `url_for` construit les URL à partir du **nom des fonctions de route**.
- Il gère aussi les paramètres dynamiques.
- Combine-le avec `redirect` pour de bonnes pratiques.

Jinja2 - Templates - Héritage

> *Dans les projets web, les applications comportent souvent plusieurs pages ayant des éléments communs, comme une navigation, un pied de page ou des styles partagés. Répéter ces éléments dans chaque fichier HTML serait non seulement fastidieux, mais rendrait aussi le projet difficile à maintenir. C'est là que l'héritage des templates de **Jinja2**, le moteur utilisé par Flask, devient essentiel.*

Cet article approfondit le concept d'héritage dans Jinja2, en expliquant ses bases, ses avantages, et en fournissant des exemples détaillés pour créer des templates dynamiques et bien structurés.

Introduction à l'héritage des templates

L'héritage permet de définir une **structure de base commune** dans un template principal (par exemple, un en-tête, une navigation, un pied de page) et de laisser les templates enfants modifier ou compléter certaines parties spécifiques (comme le contenu d'une page).

Principe fondamental

1. **Template de base** : Définit les blocs génériques, qui peuvent être personnalisés par les templates enfants.
2. **Template enfant** : Hérite de la structure du template de base, remplit les blocs et peut ajouter du contenu ou modifier les parties existantes.

Pourquoi utiliser l'héritage ?

1. **Réduction des répétitions** : Les éléments communs (entête, pied de page) sont définis une fois dans un template de base.
2. **Maintenabilité** : Une modification dans le template de base est automatiquement répercutée sur toutes les pages enfants.
3. **Organisation** : Chaque template enfant se concentre uniquement sur sa logique spécifique, en s'appuyant sur une structure commune.

Syntaxe de base de l'héritage

1. **Créer un template de base** : Utilisez des blocs Jinja2 pour définir les parties modifiables.
2. **Créer un template enfant** : Héritez du template de base avec `{% extends %}` et remplissez les blocs avec `{% block %}`.

Étape 1 : Créer un Template de Base

Le template de base définit une structure commune à toutes les pages. Par exemple, `base.html` peut inclure une en-tête, une navigation, un pied de page, et des blocs pour le contenu spécifique.

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{% block title %}Titre par défaut{% endblock %}</title>
    <link rel="stylesheet" href="/static/styles.css">
</head>
<body>
    <header>
        <h1>Mon Application</h1>
        <nav>
            <ul>
                <li><a href="/">Accueil</a></li>
                <li><a href="/about">À propos</a></li>
                <li><a href="/contact">Contact</a></li>
            </ul>
        </nav>
    </header>
    <main>
        {% block content %}
        <p>Contenu par défaut</p>
        {% endblock %}
    </main>
    <footer>
        <p>&copy; 2024 Mon Application</p>
    </footer>
</body>
</html>
```

Explications

- `{% block title %}` : Déclare un bloc modifiable pour le titre de la page.
- `{% block content %}` : Déclare un bloc modifiable pour le contenu principal.
- **Structure commune** : Les balises `<header>` et `<footer>` restent identiques pour toutes les pages.

Étape 2 : Créer un Template Enfant

Un template enfant hérite de `base.html` en utilisant `{% extends 'base.html' %}` et remplit les blocs déclarés.

▸ Exemple : Page d'accueil (`index.html`)

```
{% extends 'base.html' %}

{% block title %}Accueil{% endblock %}

{% block content %}
<h2>Bienvenue sur la page d'accueil</h2>
<p>Ceci est le contenu principal de la page d'accueil.</p>
{% endblock %}
```

▸ Exemple : Page "À propos" (`about.html`)

```
{% extends 'base.html' %}

{% block title %}À propos{% endblock %}

{% block content %}
<h2>À propos de nous</h2>
<p>Nous sommes une équipe passionnée par le développement web.</p>
{% endblock %}
```

Étape 3 : Ajouter des Blocs Optionnels

Vous pouvez déclarer des blocs optionnels dans le template de base pour des parties qui ne sont pas toujours nécessaires.

▸ Exemple avec une Barre Latérale (`base.html`)

Ajoutez un bloc pour une barre latérale :

```
<aside>
    {% block sidebar %}
    <p>Pas de contenu supplémentaire ici.</p>
    {% endblock %}
</aside>
```

Template Enfant avec Barre Latérale

```
{% extends 'base.html' %}

{% block title %}Page avec barre latérale{% endblock %}

{% block content %}
<h2>Contenu Principal</h2>
<p>Voici le contenu principal de cette page.</p>
{% endblock %}

{% block sidebar %}
<h3>Liens utiles</h3>
<ul>
    <li><a href="#">Lien 1</a></li>
    <li><a href="#">Lien 2</a></li>
</ul>
{% endblock %}
```

Étape 4 : Combiner des Blocs avec du Contenu Statique

Un template enfant peut ajouter du contenu sans remplacer entièrement un bloc.

Exemple : Ajouter du Contenu à un Bloc (`base.html`)

```
<header>
    <h1>Mon Application</h1>
    {% block header_extras %}
    {% endblock %}
</header>
```

Template Enfant qui Ajoute du Contenu

```
{% extends 'base.html' %}

{% block header_extras %}
<p>Slogan : Apprenez à coder avec nous !</p>
{% endblock %}
```

Étape 5 : Réorganiser avec des Fichiers Partiels

L'héritage peut être combiné avec des inclusions pour organiser les templates en fragments réutilisables.

▸ Créer des Fichiers Partiels

Déplacez les parties répétitives dans des fichiers partiels, par exemple, un fichier `_header.html` :

```
<header>
    <h1>Mon Application</h1>
    <nav>
        <ul>
            <li><a href="/">Accueil</a></li>
            <li><a href="/about">À propos</a></li>
            <li><a href="/contact">Contact</a></li>
        </ul>
    </nav>
</header>
```

▸ Inclure dans le Template de Base

Utilisez `{% include %}` pour insérer le fichier partiel dans `base.html` :

```
<body>
    {% include 'partials/_header.html' %}
    <main>
        {% block content %}
        <p>Contenu par défaut</p>
        {% endblock %}
    </main>
    <footer>
        <p>&copy; 2024 Mon Application</p>
    </footer>
</body>
```

Étape 6 : Gestion des Blocs Vides

Pour éviter des blocs vides non gérés, vous pouvez fournir des valeurs par défaut dans les templates de base.

Exemple : Bloc avec Valeur par Défaut

```
{% block sidebar %}
<p>Aucun contenu supplémentaire n'a été défini.</p>
{% endblock %}
```

Les templates enfants peuvent laisser ce bloc vide sans générer d'erreur.

Exemple Complet

Structure du Projet

```
projet/
│
├── templates/
│   ├── base.html
│   ├── index.html
│   ├── about.html
│   └── partials/
│       └── _header.html
│
├── static/
│   └── styles.css
│
└── app.py
```

Code Flask

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/about')
def about():
    return render_template('about.html')

if __name__ == '__main__':
    app.run(debug=True)
```

Bonnes Pratiques pour l'Héritage de Templates

1. **Structurez vos templates** :
   - Créez un fichier `base.html` avec la structure commune.
   - Utilisez des fichiers partiels pour les éléments réutilisables.
2. **Nommez vos blocs clairement** :
   - Utilisez des noms explicites pour éviter la confusion (par exemple, `content`, `sidebar`, `header_extras`).
3. **Évitez la surcharge** :
   - Ne multipliez pas les blocs inutiles dans `base.html`. Concentrez-vous sur les parties vraiment dynamiques.
4. **Testez les blocs vides** :
   - Fournissez des valeurs par défaut dans les blocs pour éviter les erreurs dans les pages enfants.

Conclusion

L'héritage des templates dans Jinja2 est une fonctionnalité puissante qui facilite la gestion de la structure et du contenu dans les applications Flask. Il améliore la réutilisabilité, réduit la répétition de code et simplifie la maintenance. En combinant héritage, inclusions, et bonnes pratiques, vous pouvez développer

Introduction à Jinja2

> *Dans cet article, nous allons apprendre à passer des données dynamiques depuis une application Flask vers des templates HTML en utilisant Jinja2. Nous commencerons par voir comment transmettre et afficher des chaînes de caractères simples, puis nous explorerons la façon de passer des listes et de les afficher de manière itérative dans nos pages web. Ces techniques vous permettront de créer des applications web interactives et dynamiques, enrichissant ainsi l'expérience utilisateur.*

Passer des Données au Template Jinja2 avec Flask

Introduction

Lorsque vous développez des applications web avec Flask, il est essentiel de savoir comment passer des données dynamiques à vos pages HTML. Flask utilise Jinja2, un moteur de templates, pour faciliter ce processus. Dans cet article, nous allons voir comment passer des chaînes de caractères et des listes aux templates Jinja2 et les afficher dynamiquement.

Étape 1 : Passer des Chaînes de Caractères au Template

Configuration de l'Application

Commencez par créer une application Flask de base. Si vous avez déjà une application, vous pouvez l'utiliser. Sinon, créez un fichier `app.py` et ajoutez le code suivant :

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    title = "Accueil"
    message = "Bienvenue sur notre site web !"
    return render_template('home.html', title=title, message=message)

if __name__ == '__main__':
    app.run(debug=True)
```

Dans ce code :

- Nous importons Flask et la fonction `render_template`.

- Nous créons une instance de l'application Flask.
- Nous définissons une route pour la page d'accueil `/`.
- Nous passons deux variables (`title` et `message`) au template `home.html` via `render_template`.

Création du Template

Ensuite, créez un dossier `templates` dans le répertoire de votre projet, puis créez un fichier `home.html` dans ce dossier avec le contenu suivant :

```
<!-- templates/home.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ title }}</h1>
    <p>{{ message }}</p>
</body>
</html>
```

Dans ce template :

- Les doubles accolades `{{ }}` sont utilisées pour insérer les valeurs des variables `title` et `message` dans le HTML.

Résultat

Lorsque vous lancez l'application en exécutant `python app.py` et que vous accédez à http://127.0.0.1:5000/, vous verrez les valeurs des variables `title` et `message` affichées sur la page web.

Étape 2 : Conditions

Dans Jinja2, le système de conditions permet d'ajuster dynamiquement le rendu des templates en fonction des valeurs de données. Ce mécanisme fonctionne de manière similaire aux structures conditionnelles dans les langages de programmation, en utilisant les blocs `{% if %}`, `{% elif %}`, et `{% else %}`. Ces blocs permettent d'exécuter des portions de code HTML spécifiques selon que des conditions soient remplies ou non. Par exemple, il est possible d'afficher un message différent selon le rôle d'un utilisateur (`admin`, `utilisateur`, etc.) ou de ne montrer certains éléments qu'en cas de disponibilité de données.

Les conditions sont particulièrement utiles pour adapter l'affichage sans avoir besoin de créer plusieurs templates. Jinja2 évalue chaque condition de haut en bas, et dès qu'une condition est vraie, le bloc correspondant est exécuté tandis que le reste est ignoré. Cette flexibilité permet de créer des interfaces plus dynamiques et personnalisées.

Modifier l'application

Ajoutez un booléen "danger":

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    title = "Accueil"
    message = "Bienvenue sur notre site web !"
    return render_template('home.html', title=title, message=message, danger=True)

if __name__ == '__main__':
    app.run(debug=True)
```

Modifier le template

```jinja
{% if danger %}
    <p class="danger">Attention, le worm a été libéré!</p>
{% else %}
    <p class="success">Tout va bien, aucune anomalie détectée.</p>
{% endif %}
```

Étape 3 : Passer et Afficher des Listes

Mise à Jour de l'Application

Modifions maintenant l'application pour passer une liste de données au template. Mettez à jour le fichier `app.py` comme suit :

```python
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    title = "Accueil"
    message = "Bienvenue sur notre site web !"
    items = ["Élément 1", "Élément 2", "Élément 3"]
    danger = True
    return render_template('home.html', title=title, message=message, items=items, danger=d

if __name__ == '__main__':
    app.run(debug=True)
```

Dans ce code :

- Nous avons ajouté une liste `items` contenant trois chaînes de caractères.
- Nous passons cette liste au template `home.html` avec les autres variables.

Mise à Jour du Template

Mettez à jour le fichier `home.html` pour afficher la liste. Ajoutez le code suivant dans le fichier :

```
<!-- templates/home.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ title }}</h1>
    <p>{{ message }}</p>

    <ul>
        {% for item in items %}
            <li>{{ item }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Dans ce template :

- Nous utilisons une boucle `for` de Jinja2 pour itérer sur chaque élément de la liste `items` et l'afficher dans une balise `<li>` à l'intérieur d'une liste non ordonnée `<ul>`.

Résultat

Lorsque vous relancez l'application en exécutant à nouveau `python app.py` et que vous accédez à http://127.0.0.1:5000/, vous verrez les éléments de la liste affichés dans une liste HTML.

Conclusion

Dans cet article, nous avons appris comment passer des données dynamiques aux templates Jinja2 dans une application Flask. Nous avons vu comment passer et afficher des chaînes de caractères ainsi que des listes en utilisant Jinja2. Ces techniques sont essentielles pour rendre vos applications web interactives et dynamiques.

Continuez à explorer Jinja2 et Flask pour découvrir des fonctionnalités plus avancées, telles que les filtres, les macros et l'héritage de templates, afin de créer des applications web encore plus puissantes et flexibles.

Projets Flask

> *Tu trouveras ici les consignes des différents projets à réaliser.*

Insertion des données (initial load)

> *Dans cet exercice, tu vas apprendre à charger des données JSON dans une base MySQL : c'est ce qu'on appelle un **initial load**. Concrètement, nous partirons d'un fichier films.json contenant une liste de films (titre, année, acteurs, genres, images, etc.) et nous allons écrire un script Python qui crée les tables nécessaires (**migrations** simples), puis importe les données de manière **idempotente** : c'est-à-dire qu'on pourra **relancer le script plusieurs fois sans créer de doublons**, tout en mettant à jour les infos si elles changent. Cet exercice te permettra de comprendre les étapes clés d'un **pipeline d'import** : créer un schéma relationnel, lire du JSON, insérer avec des contraintes, gérer les relations entre tables et vérifier le résultat avec des requêtes SQL.*

Vocabulaire

📘 Migration

En informatique (et en particulier avec les bases de données), une **migration** est une opération qui **modifie la structure de la base** afin de l'adapter aux besoins de l'application : **création** de tables, **ajout** de colonnes, ajout de **clés**, modification de types, etc.

📘 Initial Load

Un **initial load** est la toute **première opération de chargement de données** depuis une source (JSON, CSV, API…) vers une base de données ou un système cible. L'objectif est de peupler la base à partir de zéro avec un jeu complet et propre, qui servira ensuite de référence pour les mises à jour ou les ajouts incrémentaux.

📘 Idempotence

Dans ce contexte, l'**idempotence** veut dire que ton script d'import peut être relancé plusieurs fois sans créer de doublons ni abîmer la base.

Exemple concret :

- Si un film est déjà importé (même `external_id`), il ne doit pas être inséré une deuxième fois → il est simplement **mis à jour** si ses infos ont changé.
- Si un acteur ou un genre existe déjà, on le **réutilise** au lieu de créer une nouvelle ligne identique.
- Si une relation film-acteur existe déjà, on ne la recrée pas (ou bien on met à jour le rôle si nécessaire).

👉 Résultat : que tu lances l'import **1 fois ou 10 fois**, la base contient **le même état cohérent**. C'est ça, l'idempotence.

Objectif : écrire **un petit script Python** qui :

1. **crée** les tables MySQL (migrations simples),
2. **importe** le fichier `films.json`,
3. garantit l'**idempotence** (relancer le script ne duplique pas),

4. reste **compréhensible** (premier exercice).

Créé un schéma entités-associations d'abord, ainsi que la conversion en version "physique" pour MySql (tables, tables de jointure, PK's, FK's...).

Prérequis

- MySQL en local (ou Docker).
- Un utilisateur MySQL avec droits de création : `user/password`.
- Paquet Python :

```
pip install mysql-connector-python
```

- Fichier JSON : `data/films.json` (ton tableau de 25 films).

Schéma minimal (normalisé mais simple)

Nous allons créer 6 tables, mais de manière **guidée** :

- `film(id PK AI, external_id UNIQUE, titre, annee, realisateur, duree_min, resume)`
- `genre(id PK AI, nom UNIQUE)`
- `film_genre(film_id FK, genre_id FK, PK(film_id, genre_id))`
- `acteur(id PK AI, nom UNIQUE)`
- `film_acteur(film_id FK, acteur_id FK, role, PK(film_id, acteur_id))`
- `image(id PK AI, film_id FK, path)`

> ***Idempotence*** *:*
>
> - *`film.external_id` = l'`id` du JSON (UNIQUE) → on peut faire un upsert.*
> - *`genre.nom` et `acteur.nom` sont **UNIQUE** → pas de doublons.*

Étape 1 — Script “migrations simples”

But : **créer si n'existe pas** (safe à relancer).

```python
# migrate.py
import mysql.connector as mysql

DB = dict(host="127.0.0.1", user="root", password="password", database="cinema", auth_plugi

DDL = [
    # Charset/Collation pour bien gérer les accents
    "CREATE DATABASE IF NOT EXISTS cinema CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci;
    "USE cinema;",
    """
    CREATE TABLE IF NOT EXISTS film (
      id INT PRIMARY KEY AUTO_INCREMENT,
      external_id INT NOT NULL,
      titre VARCHAR(255) NOT NULL,
      annee INT NOT NULL,
      realisateur VARCHAR(255),
      duree_min INT,
      resume TEXT,
      UNIQUE KEY uq_film_external_id (external_id),
      INDEX idx_film_titre (titre),
      INDEX idx_film_annee (annee)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
    """,
    """
    CREATE TABLE IF NOT EXISTS genre (
      id INT PRIMARY KEY AUTO_INCREMENT,
      nom VARCHAR(100) NOT NULL,
      UNIQUE KEY uq_genre_nom (nom)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
    """,
    """
    CREATE TABLE IF NOT EXISTS film_genre (
      film_id INT NOT NULL,
      genre_id INT NOT NULL,
      PRIMARY KEY (film_id, genre_id),
      FOREIGN KEY (film_id) REFERENCES film(id) ON DELETE CASCADE,
      FOREIGN KEY (genre_id) REFERENCES genre(id) ON DELETE CASCADE
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
    """,
    """
    CREATE TABLE IF NOT EXISTS acteur (
      id INT PRIMARY KEY AUTO_INCREMENT,
      nom VARCHAR(255) NOT NULL,
      UNIQUE KEY uq_acteur_nom (nom)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
    """,
```

```
    """
    CREATE TABLE IF NOT EXISTS film_acteur (
      film_id INT NOT NULL,
      acteur_id INT NOT NULL,
      role VARCHAR(255),
      PRIMARY KEY (film_id, acteur_id),
      FOREIGN KEY (film_id) REFERENCES film(id) ON DELETE CASCADE,
      FOREIGN KEY (acteur_id) REFERENCES acteur(id) ON DELETE CASCADE
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
    """,
    """
    CREATE TABLE IF NOT EXISTS image (
      id INT PRIMARY KEY AUTO_INCREMENT,
      film_id INT NOT NULL,
      path VARCHAR(400) NOT NULL,
      FOREIGN KEY (film_id) REFERENCES film(id) ON DELETE CASCADE,
      INDEX idx_image_film (film_id)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
    """
]

def main():
    conn = mysql.connect(**DB)
    cur = conn.cursor()
    for stmt in DDL:
        cur.execute(stmt)
    conn.commit()
    cur.close()
    conn.close()
    print("✅ Migrations OK (tables prêtes).")

if __name__ == "__main__":
    main()
```

À comprendre

- `CREATE TABLE IF NOT EXISTS` → relancer sans casser.
- Clés **UNIQUE** + **FK** → cohérence + idempotence.

Étape 2 — Import (lecture JSON → insert idempotent)

Stratégie **simple** :

- Upsert du film via `external_id UNIQUE` (**sans dupliquer**).
- `get_or_create` pour `genre` et `acteur` (sur leur `nom UNIQUE`).
- Récréer les liaisons (film-genre, film-acteur) **sans dupliquer**.

```python
# import_films.py
import json
import mysql.connector as mysql

DB = dict(host="127.0.0.1", user="root", password="password", database="cinema", auth_plugi

def get_conn():
    return mysql.connect(**DB)

def get_or_create_genre(cur, nom):
    # essaie d'insérer, sinon récupère l'id existant
    cur.execute("INSERT IGNORE INTO genre(nom) VALUES (%s)", (nom,))
    cur.execute("SELECT id FROM genre WHERE nom=%s", (nom,))
    return cur.fetchone()[0]

def get_or_create_acteur(cur, nom):
    cur.execute("INSERT IGNORE INTO acteur(nom) VALUES (%s)", (nom,))
    cur.execute("SELECT id FROM acteur WHERE nom=%s", (nom,))
    return cur.fetchone()[0]

def upsert_film(cur, f):
    """
    Retourne l'id interne du film après upsert.
    ON DUPLICATE KEY UPDATE garantit l'idempotence.
    """
    cur.execute("""
        INSERT INTO film (external_id, titre, annee, realisateur, duree_min, resume)
        VALUES (%s, %s, %s, %s, %s, %s)
        ON DUPLICATE KEY UPDATE
          titre=VALUES(titre),
          annee=VALUES(annee),
          realisateur=VALUES(realisateur),
          duree_min=VALUES(duree_min),
          resume=VALUES(resume)
    """, (f["id"], f["titre"], f["annee"], f.get("realisateur"), f.get("duree_min"), f.get(
    # Récupérer l'id interne
    cur.execute("SELECT id FROM film WHERE external_id=%s", (f["id"],))
    return cur.fetchone()[0]

def link_film_genre(cur, film_id, genre_id):
    # évite les doublons grâce à la PK(film_id, genre_id)
    cur.execute("""
        INSERT IGNORE INTO film_genre (film_id, genre_id) VALUES (%s, %s)
    """, (film_id, genre_id))

def link_film_acteur(cur, film_id, acteur_id, role):
```

```
    cur.execute("""
        INSERT INTO film_acteur (film_id, acteur_id, role)
        VALUES (%s, %s, %s)
        ON DUPLICATE KEY UPDATE role=VALUES(role)
    """, (film_id, acteur_id, role))

def insert_images(cur, film_id, images):
    # simple: on ajoute; si tu veux l'idempotence stricte, vérifie l'existence du path avan
    for p in images:
        cur.execute("INSERT INTO image (film_id, path) VALUES (%s, %s)", (film_id, p))

def import_json(path_json):
    with open(path_json, encoding="utf-8") as f:
        films = json.load(f)

    conn = get_conn()
    cur = conn.cursor()
    try:
        # petite transaction globale (ici dataset modeste)
        conn.start_transaction()
        for f in films:
            # 1) upsert film
            film_id = upsert_film(cur, f)

            # 2) genres
            for g in f.get("genres", []):
                gid = get_or_create_genre(cur, g.strip())
                link_film_genre(cur, film_id, gid)

            # 3) acteurs
            for a in f.get("acteurs", []):
                aid = get_or_create_acteur(cur, a["nom"].strip())
                role = a.get("role")
                link_film_acteur(cur, film_id, aid, role)

            # 4) images (option simple : on ne déduplique pas ici)
            insert_images(cur, film_id, f.get("images", []))

        conn.commit()
        print(f"✅ Import OK ({len(films)} films).")
    except Exception as e:
        conn.rollback()
        print("❌ Erreur, rollback :", e)
    finally:
        cur.close()
        conn.close()
```

```
if __name__ == "__main__":
    import_json("data/films.json")
```

Points pédagogiques clés

- `INSERT IGNORE` + `UNIQUE` → **get_or_create** sans si/else compliqué.
- `ON DUPLICATE KEY UPDATE` → **upsert** propre.
- **Transaction** : si ça casse, **rollback** (rien d'à moitié importé).
- **Idempotence** : relance le script → pas de doublons, les films sont **mis à jour**.

Étape 3 — Vérifications (exemples concrets)

A) Compter les films importés

```
SELECT COUNT(*) FROM film;
```

B) Vérifier un film précis

```
SELECT id, external_id, titre, annee FROM film WHERE external_id=101;
```

C) Genres liés à "Inception"

```
SELECT g.nom
FROM film f
JOIN film_genre fg ON fg.film_id=f.id
JOIN genre g ON g.id=fg.genre_id
WHERE f.external_id=101;
```

D) Acteurs + rôle

```
SELECT a.nom, fa.role
FROM film f
JOIN film_acteur fa ON fa.film_id=f.id
JOIN acteur a ON a.id=fa.acteur_id
WHERE f.external_id=101;
```

E) Images liées

```
SELECT path FROM image i
JOIN film f ON f.id=i.film_id
WHERE f.external_id=101;
```

Bonnes pratiques (version “élève”)

- **Toujours** mettre `utf8mb4` pour gérer accents/emoji.
- Normaliser strings : `strip()`, éviter espaces doubles.
- **Ne jamais** mettre d’URL en dur côté Flask : stocke juste le **path** d’image.
- **Commence petit** : teste d’abord avec 3 films, puis 25.
- Si tu veux que les images soient aussi **idempotentes**, remplace `insert_images` par :

```
cur.execute("SELECT 1 FROM image WHERE film_id=%s AND path=%s", (film_id, p))
if not cur.fetchone():
    cur.execute("INSERT INTO image (film_id, path) VALUES (%s, %s)", (film_id, p))
```

Variante “ultra-simplifiée” (si nécessaire)

> *Pour une première heure de cours : **2 tables** seulement.*

- `film(id PK AI, external_id UNIQUE, titre, annee, resume)`
- `image(id PK AI, film_id FK, path)`

Idées : ignorer dans un premier temps `genre`, `acteur`, puis **ajouter** ces tables à l’exercice 2 (migration qui étend le schéma, et script d’import qui les remplit).

Ce que tu dois réussir à la fin

- Lancer `python migrate.py` → “✅ Migrations OK”.
- Lancer `python import_films.py` → “✅ Import OK (25 films)”.
- Relancer `python import_films.py` → **aucun doublon**, mais les valeurs mises à jour si tu modifies le JSON.
- Vérifier 2–3 films avec les requêtes SQL de contrôle.

Prochaines étapes (facultatif)

- Ajouter un **mode** `--dry-run` qui affiche les actions sans écrire.
- Écrire des **tests** (pytest) pour `get_or_create_*`.
- Ajouter des **indexes** selon les recherches que tu feras dans Flask (`WHERE annee=…` , `LIKE titre`).

Si tu veux, je te fournis une **version monofichier** qui fait *migrations + import* en un seul `python app.py` (avec un petit menu texte).

Intégration d'un template existant

> *Sans SQL, juste du json pour le plaisir*

Exercice de révision Flask

Objectifs

- Lire des données depuis un fichier **JSON**.
- Créer **3 pages** avec Flask : **accueil**, **liste**, **détail** (route dynamique).
- Utiliser **Jinja2** pour boucler, formater et lier les pages avec `url_for`.
- **Utiliser et adapter un template existant**

Énoncé (commun à tous les thèmes)

Tu choisis **un** des 4 thèmes ci-dessous et tu réalises une mini-appli Flask avec **3 pages** :

Accueil ("Dashboard") `/`

- Sur cette page, tu vas implémenter un dashboard simplifié:
- Phrase d'intro + **lien** vers la page liste.
- **Compteur** d'éléments (ex: "12 recettes").
- **Élément mis en avant** (choisi au hasard).
- Optionnel : un champ de recherche qui redirige vers la liste `?q=...`.



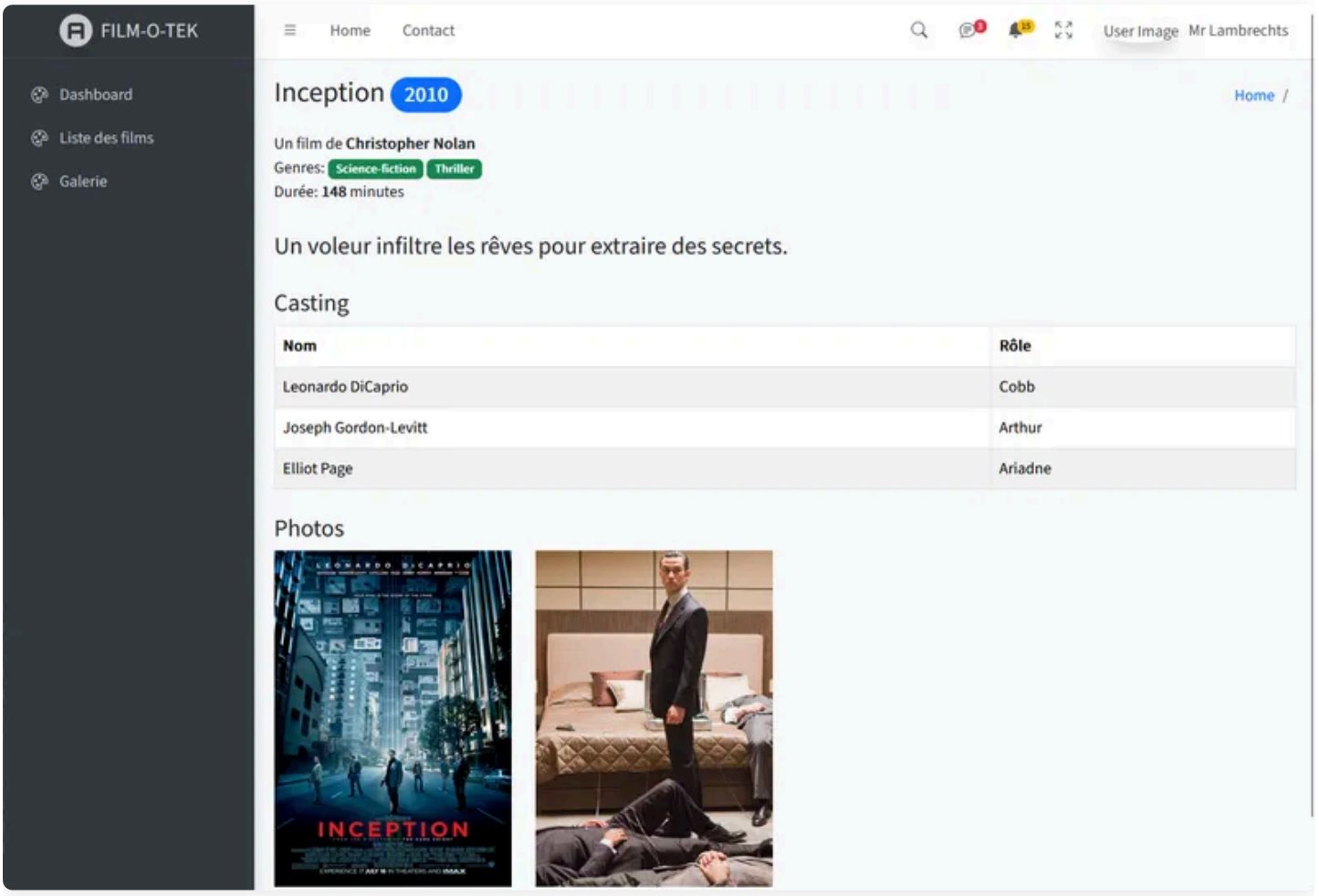
Liste `/<prefix>` (ex: `/recettes`, `/films`)

- Lecture du fichier JSON → affichage d'une **liste** (table ou cartes).
- Chaque élément a un **lien** vers sa page **détail** via son `id`.

- Optionnel : filtre texte `?q=` sur un ou deux champs (ex: titre, nom).



- **Détail** `/<prefix>/<int:id>` (ex: `/recettes/42`)
- Affiche **toutes** les infos de l'élément choisi.
- Si l'`id` n'existe pas : **404**.



- **Contraintes techniques & bonnes pratiques**
- **Structure conseillée**

```
app.py
data/<theme>.json
templates/
  base.html
  index.html
  list.html
  detail.html
static/style.css (optionnel)
```

- **Lecture JSON** : `json.load(open('data/<theme>.json', encoding='utf-8'))`
- **Routage** : `@app.route('/<prefix>/<int:item_id>')`
- **Recherche** : `request.args.get('q', '')` (optionnel)
- **Jinja2** : `extends "base.html"`, boucles `{% for ... \%}`, `{{ url_for('detail', item_id=... ) }}`
- **404 propre** : `abort(404)` si non trouvé.
- **Aucune écriture** dans le JSON (lecture seule).

▸ Utiliser un admin template “prêt à l’emploi” (AdminLTE)

Pour réaliser ce projet, tu vas utiliser un *admin template* existant: https://adminlte.io/.

Le principe est "simple": il s'agit d'un template "prêt à l'emploi", avec des pages et des *widgets* réutilisables. Ton objectif: à partir de la documentation et des exemples/démos, tu vas "découper" le template (identifier les composants réutilisables) pour en récupérer le HTML nécessaire et l'intégrer dans ton site.

Pourquoi / avantages

- **Gain de temps** : navigation, header, sidebar, cartes, tableaux, formulaires, modales… tout est déjà *designé* et réutilisable.
- **Design cohérent et responsive** (AdminLTE v4 est basé sur Bootstrap 5) : tu te concentres sur la logique Flask/Jinja plutôt que sur le CSS.
- **Écosystème riche** : widgets, icônes, charts, tables avancées (DataTables), thèmes sombres/clairs.
- **Maintenance facilitée** : un seul layout `base.html` et des composants Jinja = moins de duplication.

Comment s’y prendre (version rapide)

1. **Installer les assets**
   - **CDN** (le plus simple pour démarrer) : inclure CSS/JS d’AdminLTE, Bootstrap, Font Awesome dans `base.html`.
   - **Local** (prod/écoles sans Internet) : télécharge le bundle et place-le dans `static/` (ex. `static/adminlte/…`).
2. **Créer un layout global** `templates/base.html`
   - Structure AdminLTE : `<body class="layout-fixed">`, **navbar**, **sidebar**, **content-wrapper**, **footer**.
   - Définis des blocs Jinja : `block title`, `block content`, `block scripts` pour injecter le contenu des pages.
3. **Étendre le layout dans tes pages**

- `index.html`, `list.html`, `detail.html` → `{% extends "base.html" %}` + contenu dans `block content`.

4. **Composer avec des “partials”**
   - Découpe `navbar.html`, `sidebar.html`, `footer.html` et `include`-les dans `base.html` pour réutiliser facilement.
5. **Intégrer les composants**
   - Utilise les classes AdminLTE (cards, badges, tables, forms) dans tes templates Jinja.
   - Pour des listes dynamiques : boucle `{% for item in items %}` dans une **card** ou **table** AdminLTE.
6. **Bonnes pratiques**
   - Centralise les liens CSS/JS dans `base.html`.
   - Évite de mélanger d’autres frameworks CSS pour limiter les conflits.
   - Crée un `app.css` (dans `static/`) pour tes surcharges légères, pas de gros hacks.

Mini-exemple de squelette `base.html` (CDN)

```
<!doctype html>
<html lang="fr">
<head>
  <meta charset="utf-8">
  <title>{% block title %}Mon app{% endblock %}</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <!-- Bootstrap 5 + AdminLTE 4 (CDN d’exemple) -->
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5/dist/css/bootstrap.
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/admin-lte@4/dist/css/adminlte.m
  <link rel="stylesheet" href="{{ url_for('static', filename='app.css') }}">
</head>
<body class="layout-fixed">
  {% include "partials/navbar.html" %}
  <div class="wrapper">
    {% include "partials/sidebar.html" %}
    <div class="content-wrapper p-3">
      {% block content %}{% endblock %}
    </div>
  </div>

  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5/dist/js/bootstrap.bundle.min.js"></
  <script src="https://cdn.jsdelivr.net/npm/admin-lte@4/dist/js/adminlte.min.js"></script>
  {% block scripts %}{% endblock %}
</body>
</html>
```

À retenir

- Un template admin te fournit une **architecture visuelle prête**, accélère le dev et uniformise tes pages.

- Commence par **intégrer le layout**, puis **étends**-le dans tes vues Flask avec Jinja.
- Reste léger sur les surcharges CSS et garde une **hiérarchie de templates** propre (base → partials → pages).

Check rapide (auto-éval)

- [ ] Accueil affiche un **compteur** et un lien vers la liste
- [ ] Liste affiche **tous** les éléments avec liens vers détail
- [ ] Détail par **id** (route dynamique) + 404 si id inconnu
- [ ] Utilisation de `url_for` partout (pas d'URL en dur)
- [ ] Code lisible, templates factorisés avec `base.html`

Thèmes (choisis 1) + fichier JSON correspondant

Thème A — Recettes de cuisine

- **Prefix** : `/recettes`
- **Idées d'affichage** Liste : nom, catégorie, temps. Détail : ingrédients (liste), étapes (liste), indicateur végétarien.

data/recettes.json

```
[
  {
    "id": 1,
    "nom": "Pâtes carbonara",
    "categorie": "Plat",
    "temps_min": 25,
    "vegetarien": false,
    "ingredients": ["Spaghetti", "Œufs", "Pancetta", "Parmesan", "Poivre"],
    "etapes": [
      "Cuire les pâtes al dente",
      "Dorer la pancetta",
      "Mélanger œufs + parmesan",
      "Égoutter, hors du feu incorporer la sauce",
      "Poivrer et servir"
    ],
    "images": [
      "/static/img/recettes/1_1.jpg",
      "/static/img/recettes/1_2.jpg"
    ]
  }
]
```

▸ Thème B — Bibliothèque de films

- **Prefix** : `/films`
- **Idées d'affichage** Liste : titre, année, genres. Détail : résumé, réalisateur, durée.

data/films.json

```
[
  {
    "id": 101,
    "titre": "Inception",
    "annee": 2010,
    "realisateur": "Christopher Nolan",
    "duree_min": 148,
    "genres": ["Science-fiction", "Thriller"],
    "resume": "Un voleur infiltre les rêves pour extraire des secrets.",
    "acteurs": [
      {"nom": "Leonardo DiCaprio", "role": "Cobb"},
      {"nom": "Joseph Gordon-Levitt", "role": "Arthur"},
      {"nom": "Elliot Page", "role": "Ariadne"}
    ],
    "images": [
      "/static/img/films/101_poster.jpg",
      "/static/img/films/101_scene1.jpg"
    ]
  }
]
```

▸ Thème C — Événements Quiz (type "Quiz Factory")

- **Prefix** : `/events`
- **Idées d'affichage** Liste : titre, date, ville, thème. Détail : description, lieu, prix indicatif.

data/events.json

```
[
  {
    "id": 2001,
    "titre": "Quiz Musical – Années 90",
    "date": "2025-09-20",
    "ville": "Bruxelles",
    "lieu": "Salle Delta",
    "theme": "Musique",
    "prix_eur": 8.5,
    "description": "Un quiz musical convivial autour des tubes des années 90.",
    "categories": [
      {"nom": "Blind Test", "nb_questions": 10},
      {"nom": "Paroles manquantes", "nb_questions": 8},
      {"nom": "Covers & Remix", "nb_questions": 6}
    ],
    "images": [
      "/static/img/events/2001_affiche.jpg",
      "/static/img/events/2001_scene.jpg"
    ]
  }]
```

Thème D — CanSat — Équipes & Missions

- **Prefix** : `/cansat`
- **Idées d'affichage** Liste : équipe, mission secondaire, capteurs. Détail : masse, fréquence radio (texte), notes.

data/cansat.json

```
[
  {
    "id": 501,
    "equipe": "AstroPico",
    "mission_secondaire": "Cartographier température & humidité",
    "capteurs": ["DHT22", "BMP280", "GPS"],
    "masse_g": 280,
    "radio": "LoRa 433 MHz (démo), logs sur carte SD",
    "notes": "Priorité à la robustesse mécanique.",
    "membres": [
      {"prenom": "Lina", "role": "Chef.fe d'équipe"},
      {"prenom": "Yanis", "role": "Électronique/Capteurs"},
      {"prenom": "Zoé", "role": "Traitement des données"}
    ],
    "images": [
      "/static/img/cansat/501_proto.jpg",
      "/static/img/cansat/501_equipe.jpg"
    ]
  }
]
```

Indices techniques (si tu bloques)

- Charger les données **une fois** au démarrage (module-level) ou à chaque requête si tu préfères la simplicité :

```python
import json
with open('data/recettes.json', encoding='utf-8') as f:
    RECETTES = json.load(f)
```

- Retrouver un élément par `id` :

```python
item = next((x for x in RECETTES if x["id"] == item_id), None)
```

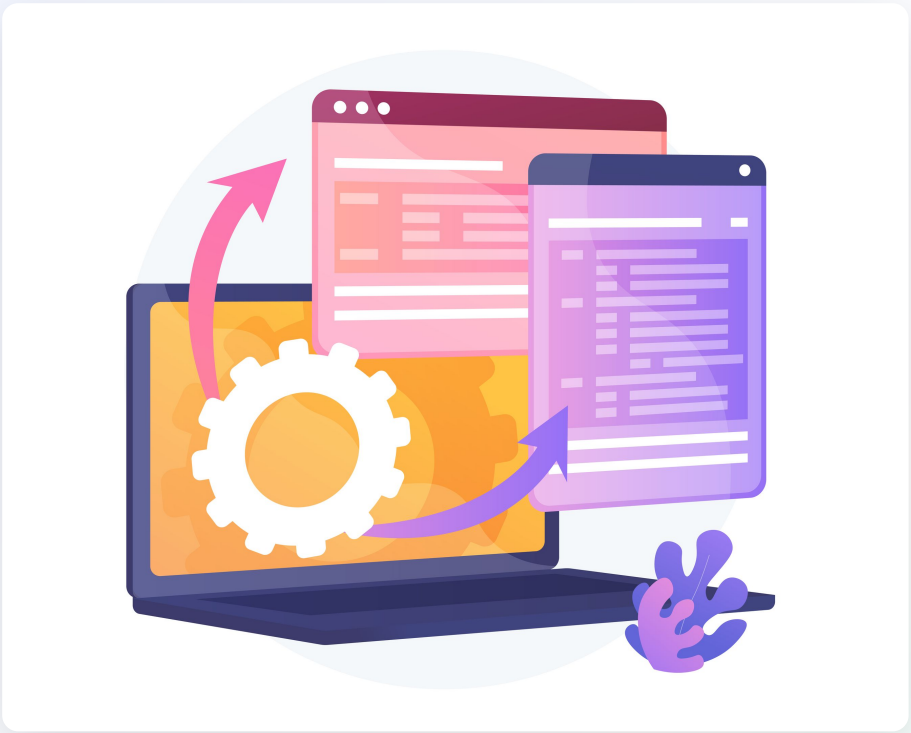
  puis `abort(404)` si `item` est `None`.

- Lier vers le détail en Jinja :

```
<a href="{{ url_for('detail_recette', item_id=recette.id) }}">{{ recette.nom }}</a>
```

Extensions (si tu as de l’avance)

- Tri par `?sort=` (ex: `annee`, `nom`), pagination simple `?page=2`.
- Champ **slug** dans le JSON et route `/.../<int:id>-<slug>`.
- Page **404 personnalisée** et base de styles CSS minimaliste.



SYSTÈMES D'EXPLOITATION

Fichiers et dossiers

> *Un **fichier** est un **ensemble de données** (texte, image, vidéo, programme, …) enregistré sur un support de stockage (disque dur, clé USB, carte mémoire…). Un **dossier** (ou répertoire) sert à **organiser les fichiers***

Objectifs

- Comprendre ce qu'est un fichier et un dossier.
- Savoir lire la structure d'un disque dur (arborescence).
- Faire la différence entre chemin absolu et chemin relatif.

Qu'est-ce qu'un fichier ?

Un fichier (GB ***file***) est un **ensemble de données** (texte, image, vidéo, programme, …) enregistré sur un support de stockage (disque dur, clé USB, carte mémoire…).

Chaque fichier a :

- **Un nom** (ex. `cours`)
- **Une extension** (ex. `.docx`, `.jpg`, `.mp3`) qui indique son type.

Le nom + l'extension donnent le **nom de fichier complet**.

Nom de fichier complet = [NOM.DU.FICHIER].[EXT]

Exemples :

- Nom complet: `rapport.docx` → nom de fichier = `rapport`, extension = `docx`
- Nom complet: `photo.jpg` → nom de fichier = `photo`, extension = `jpg`

Le nom de fichier est tout ce qui se trouve **AVANT le dernier point** du nom de fichier complet.

L'extension du fichier se trouve **APRÈS le dernier point** du nom du fichier. Elle indique le type de fichier et permet de déterminer le programme pour l'ouvrir. Par exemple, .docx pour un document Word ou .jpg pour une image.

Exemple:

Qu'est-ce qu'un dossier ?

Un dossier (GB ***folder***), aussi appelé répertoire (GB ***directory***) sert à **organiser les fichiers**.

- Un dossier peut contenir **des fichiers** et **d'autres dossiers**.
- On parle de **dossier parent** pour les dossiers qui contiennent d'autres dossiers.

- On parle de **dossier enfant (ou sous-dossier)** pour les dossiers qui sont à l'intérieur d'un autre.

💡 En résumé : **les dossiers contiennent des fichiers et parfois d'autres dossiers**.

Arborescence des dossiers

Les dossiers sont organisés comme un **arbre** :

- En haut, on trouve la **racine** (GB ***root***), qui est le point de départ, le dossier "le plus en haut".
- Chaque dossier peut contenir d'autres dossiers ou fichiers.

Exemple sous Windows :

```
C:\
 └── Utilisateurs
      └── Adrien
      |    └── Documents
      |    |    └── notes.txt
      |    └── Téléchargements
      └── liste.txt
```

Ici :

- `C:\` est la racine du disque C.
- `Utilisateurs` est un enfant de `C:\`.
- `Utilisateurs` a 2 enfants:
  - le dossier `Adrien`
  - le fichier `liste.txt`
- `Adrien` est l'enfant de `Utilisateurs`.
- `Adrien` a 2 dossiers enfants:
  - `Documents` est l'enfant de `Adrien`.
  - `Téléchargements` est l'enfant de `Adrien`.
- `Adrien` est donc le parent de `Documents` et `Téléchargements`
- `notes.txt` est un fichier dans `Documents`.

Les chemins d'accès

Un **chemin** indique où se trouve un fichier ou dossier.

Chemin absolu

C'est le chemin **complet** depuis la racine. Exemple :

```
C:\Utilisateurs\Adrien\Documents\notes.txt
```

▸ Chemin relatif

C'est le chemin **par rapport à l'endroit où l'on se trouve**.

- `..` signifie « remonter d'un dossier ».

Exemple : Si je suis dans `C:\Utilisateurs\Adrien` et que je veux lire le fichier `notes.txt` qui se trouve dans `Documents`, je peux écrire :

```
type Documents\notes.txt
```

Si je veux remonter d'un cran pour afficher le fichier `liste.txt` dans `C:\Utilisateurs`, je peux écrire :

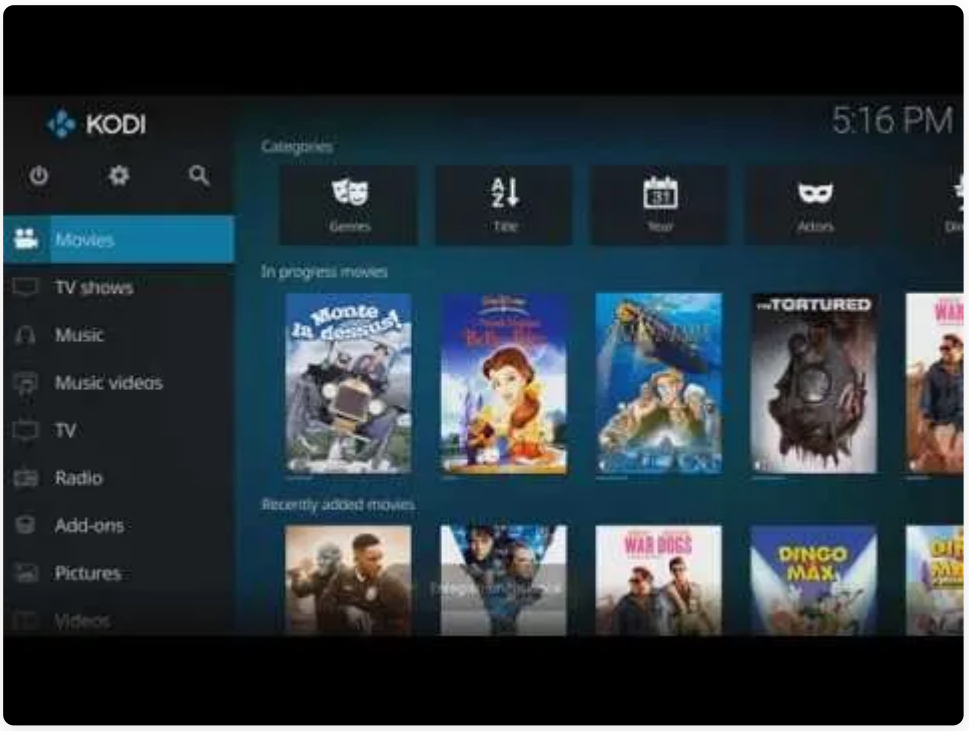
```
type ..\liste.txt
```

À retenir

- Un fichier = données enregistrées avec un nom et une extension.
- Un dossier contient des fichiers et/ou d'autres dossiers.
- Organisation en **arborescence** (racine → parents → enfants).
- **Chemin absolu** = depuis la racine.
- **Chemin relatif** = depuis l'endroit où tu es déjà (`..` pour remonter).

Organisez votre vidéothèque (2025)

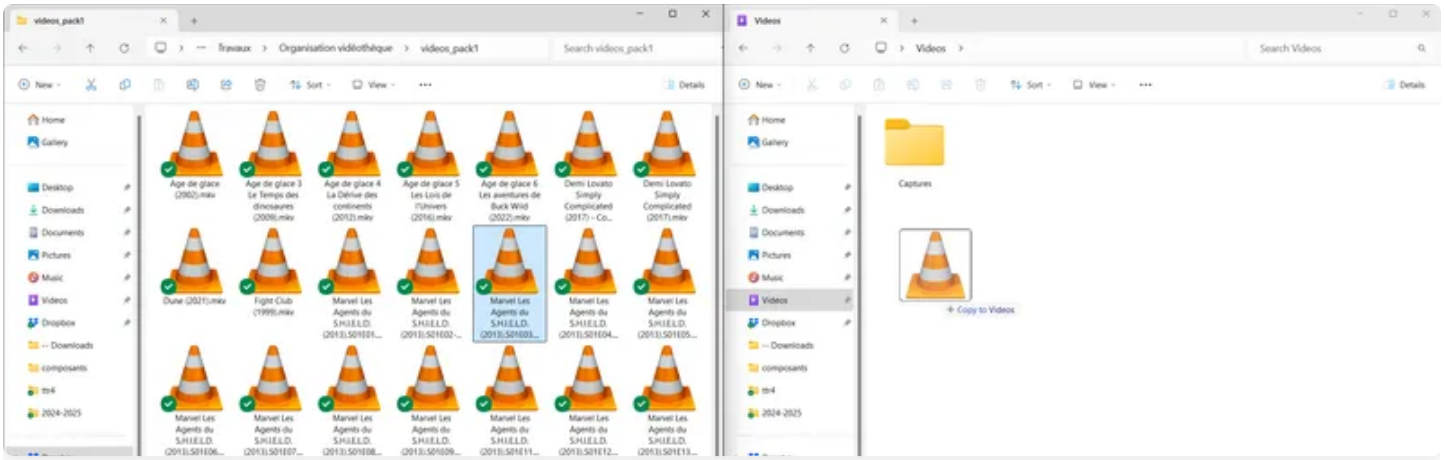
Dans cet exercice, tu vas apprendre à organiser une collection de fichiers vidéo en créant une arborescence de dossiers logique et structurée à l'aide de l'Explorateur de fichiers. Tu devras trier des fichiers, les déplacer, les renommer et les ranger de façon méthodique en suivant les consignes. Tu auras également l'occasion d'utiliser des fonctionnalités de l'Explorateur, comme le 'glisser-déposer' (drag & drop), et tu découvriras comment utiliser deux fenêtres simultanément pour travailler plus efficacement.



Objectifs d'Apprentissage

À la fin de cet exercice, tu seras capable de :

- Organiser des fichiers et des dossiers sur un disque.
- Analyser et comprendre un article technique décrivant une structure de fichiers.
- Créer des dossiers à l'aide de l'explorateur de fichiers.
- Sélectionner, copier, déplacer et renommer des dossiers et des fichiers.
- Utiliser deux fenêtres de l'explorateur de fichiers simultanément pour travailler plus efficacement.



En bonus :

- La méthode de rangement pourra t'aider à créer ton propre centre multimédia à la maison.

Instructions

Pour te permettre de suivre ta progression, utilise le `Watcher` fourni dans l'archive (voir instructions en bas de page).

1. **Télécharge ton fichier de travail sur fichiers.ttrinfo.be**
2. **Décompresse l'archive** : Une fois téléchargée, décompresse l'archive sur ton disque dans un dossier de ton choix. ⚠️ Vérifie ensuite que tu travailles dans le dossier extrait, et PAS dans l'archive.
3. **Création de l'arborescence de dossiers** : En suivant les recommandations ci-dessous, crée une arborescence de dossiers qui te permettra d'organiser ta vidéothèque. Voici quelques conseils pour la création des dossiers :
   - Tu peux créer tous les dossiers en une seule fois, puis y déplacer les fichiers.
   - Ou, tu peux les créer au fur et à mesure que tu classes tes fichiers.
   - Mélange les deux méthodes si tu préfères. Trouve la stratégie qui te convient le mieux !

Astuce: tu peux aussi y aller progressivement, niveau par niveau. Par exemple, tu peux d'abord déplacer tous les films dans le dossiers `Films` et toutes les séries dans le dossier `Séries` (niveau 1).

Puis pour les films, passe au niveau 2: crée les dossiers des sagas et déplaces-y les films correspondants. Crée les dossiers de chaque film et déplace le fichier correspondant.

Puis pour les séries, passe au niveau 2: crée un dossier par série et déplces-y tous les fichiers de la série. Ensuite, au 3e niveau, crée les dossiers des saisons et déplaces-y les fichiers correspondants.

🤓 Utilise deux fenêtres de l'Explorateur de fichiers : ce sera beaucoup plus simple pour glisser-déposer tes documents d'un endroit à l'autre.

Imagine que ton ordinateur est comme ta chambre : tu as une caisse à gauche et une caisse à droite. Tu prends des objets dans la première et tu les déplaces dans la seconde. Avec deux explorateurs ouverts, c'est exactement la même chose, mais pour organiser tes fichiers.

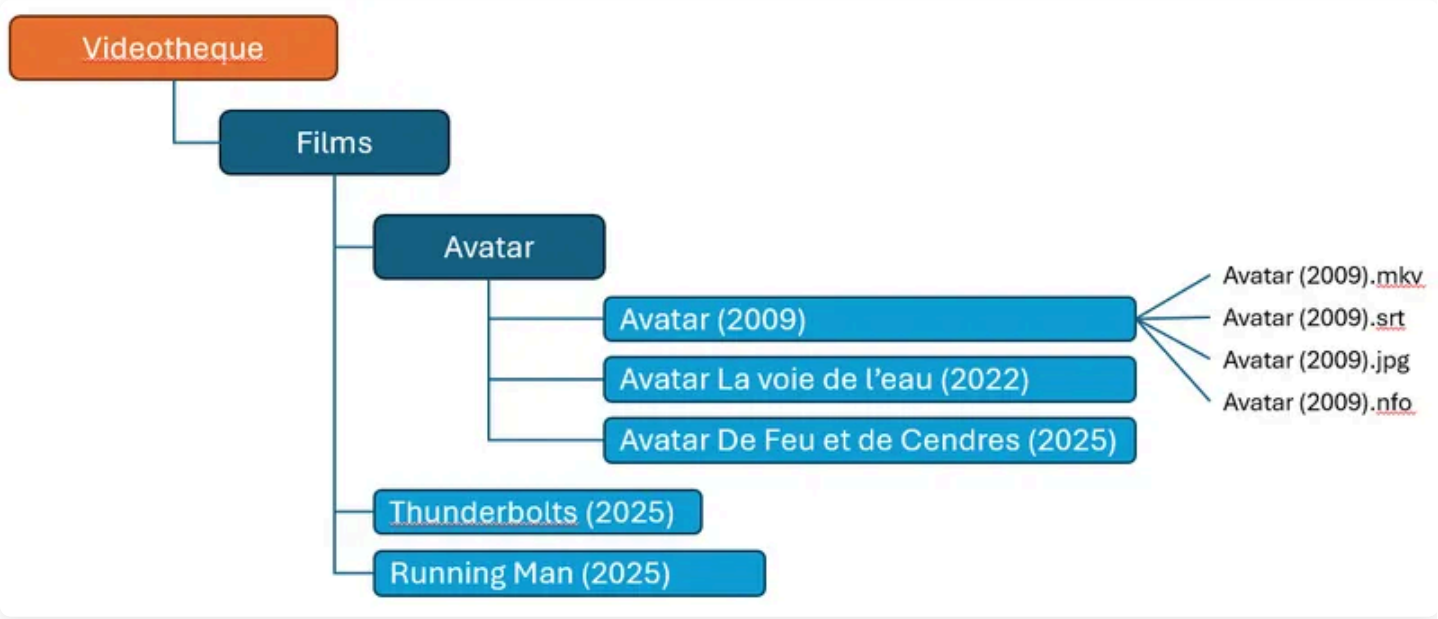
1. **Classement des fichiers** : Utilise le "glisser-déposer" (drag & drop) pour déplacer les fichiers dans les dossiers appropriés. Pour plus de rapidité, ouvre deux fenêtres de l'Explorateur de fichiers : une pour l'arborescence de dossiers que tu crées, et l'autre pour les fichiers que tu veux trier.

2. **Besoin d'aide ?** : Consulte tes notes de cours sur l'utilisation de l'Explorateur de fichiers. N'oublie pas que ton professeur est également là pour répondre à tes questions si nécessaire.

Consignes

Ces consignes reposent sur **le bon sens et des conventions** largement acceptées dans la communauté des utilisateurs de médiathèques. L'idée principale est de séparer les différents types de médias (films, séries, concerts, etc.) dans des dossiers distincts, puis de les organiser à l'intérieur selon des critères spécifiques (par exemple, les films d'une même saga regroupés dans un dossier, les séries classées par saisons). Cette méthode te permettra non seulement de retrouver facilement tes fichiers à la main, mais aussi d'**optimiser leur reconnaissance par des logiciels comme Kodi**.

Les films



Les films doivent se trouver dans le dossier `Films`.

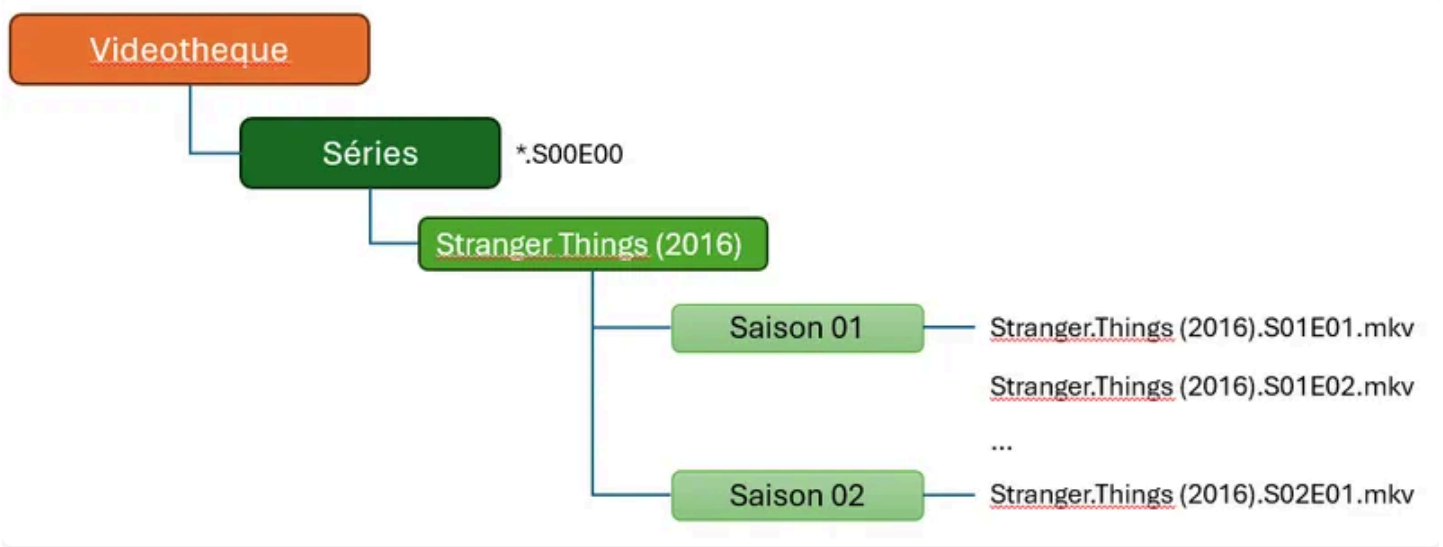
S'il n'y a qu'un seul film, place-le dans son propre dossier (nom identique à celui du film). Ex: `Thundebolts (2025).mkv` --> dossier `Thundebolts (2025)`.

S'il y a plusieurs films d'une même "série", il s'agit d'une **saga**. Dans ce cas les films doivent déplacés **dans un répertoire qui porte le nom de la saga**. Exemples de saga: Avatar, Hunger Games...

Tous les fichiers en rapport avec un même film doivent se trouver dans le même dossier.

Les séries

Tu peux identifier une série à son nom de fichier qui se termine par S00E00 --> S00 représente le numéro de saison et E00 le numéro de l'épisode dans la saison.



Chaque série doit être rangée dans son propre dossier.

Dans le dossier d'une série, il faut créer un dossier pour chaque saison, qui contient tous les fichiers des épisodes de la saison. Contrairement aux films, pas besoin de créer un dossier pour chaque épisode.

Suivi de progression

Pour suivre ta progression en temps réel, lance le programme `Watcher.exe` qui se trouve dans ton archive. Celui-ci va vérifier automatiquement si les fichiers sont à la bonne place et envoyer tes statistiques sur un serveur.

Le classement est affiché en classe.

IMPORTANT: Le classement sert à créer une petite compétition sympathique entre vous. Mais le plus important, c'est ceci : **si tu n'es pas encore en haut du classement, cela signifie simplement que tu es en train d'apprendre. Et c'est une très bonne nouvelle, parce que le but de l'activité, justement, c'est de progresser. Chaque place gagnée montre que tu avances et que tes efforts paient**. Et à la fin, ce qui compte, c'est que tu es parti·e de 0 fichiers classés. Quelle belle progression!

Et si tu as terminé l'exercice, va expliquer comment faire à quelqu'un qui a besoin d'aide!

Raccourcis claviers utiles

- `F2` : renommer un fichier
- `CTRL-SHIFT-N` : créer un nouveau dossier
- `Win- →` : déplacer la fenêtre vers la droite
- `Win- ←` : déplacer la fenêtre vers la gauche

Séquence utile pour créer le dossier d'un film/d'une série:

- `F2` (renomme)
- `CTRl-C` (copier le nom sélectionné, sans l'extension)
- `ESC` ou `ECHAP` ou `ENTRÉE` (qui le mode sélection)
- `CTRL-SHIFT-N` (créer un nouveau dossier)
- `CTRL-V` coller le nom du fichier

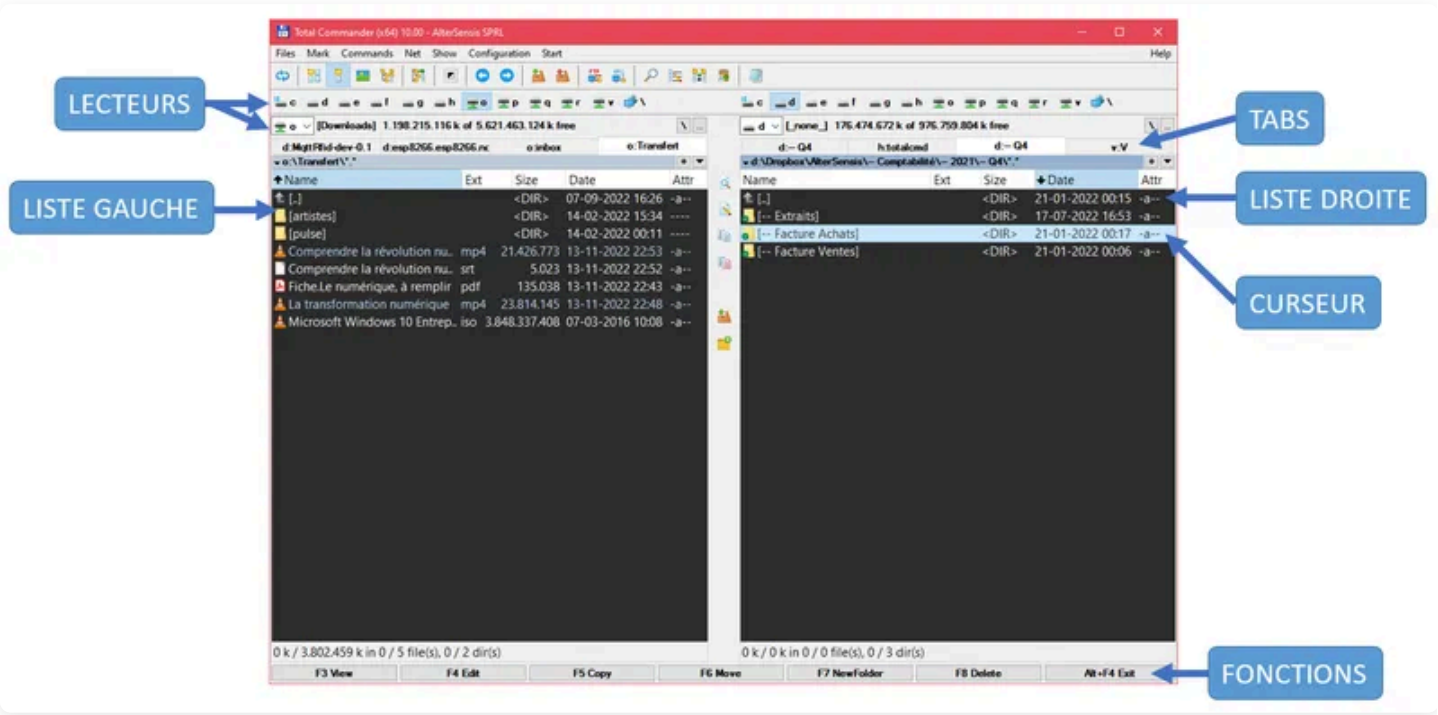
- `ENTRÉE` valider la création du dossier

Les outils Windows

Total Commander

> *Dans les métiers de l'informatique, l'efficacité et la précision sont essentielles : manipuler des fichiers, comparer des dossiers, automatiser des tâches, naviguer rapidement dans une arborescence complexe… L'explorateur de fichiers de Windows suffit pour un usage quotidien, mais il atteint vite ses limites.*

Total Commander offre une alternative professionnelle, optimisée pour la productivité, largement utilisée par les administrateurs système, techniciens et développeurs.



Qu'est-ce que Total Commander ?

Total Commander est un **gestionnaire de fichiers avancé** conçu pour Windows. Il repose sur une interface à **double panneau** (gauche/droite), permettant de visualiser simultanément deux dossiers. C'est l'héritier moderne des gestionnaires de fichiers de type "Commander", très populaires dans les environnements DOS et Linux.

Il permet notamment de :

- naviguer dans les dossiers rapidement,
- déplacer ou copier des fichiers,
- gérer les archives (ZIP, RAR, 7Z…),
- synchroniser des dossiers,
- renommer des fichiers en masse,
- se connecter à des serveurs FTP/SFTP,
- étendre les fonctionnalités par des plugins.

À quoi sert Total Commander ?

Gérer les fichiers plus efficacement

Grâce au double panneau, chaque opération (copie, déplacement, création de dossier) devient beaucoup plus rapide. Sélection → F5 → Copie. C'est plus simple et plus efficace qu'avec l'explorateur Windows.

Automatiser ou accélérer les tâches répétitives

Total Commander propose :

- renommage massif avec règles,
- sélection intelligente de fichiers,
- favoris pour sauter immédiatement d'un dossier à l'autre,
- recherche avancée avec filtres et expressions régulières.

Travailler avec des archives comme si c'étaient des dossiers

ZIP, RAR, 7z… On peut ouvrir une archive, explorer son contenu et manipuler ce qu'elle contient **sans logiciel supplémentaire**.

Synchroniser des répertoires

Idéal pour les informaticiens : on compare deux dossiers, repère les différences, et synchronise en quelques clics.

Accéder à des serveurs externes

Un module intégré permet d'accéder à :

- FTP,
- FTPS,
- SFTP (via plugin).

Pratique pour administrer des sites ou des serveurs distants.

Pourquoi Total Commander est-il supérieur à l'explorateur Windows ?

Double panneau = vitesse incomparable

L'explorateur Windows nécessite plusieurs manipulations pour passer d'un dossier à un autre ou copier un fichier. Ici, tout se fait en un geste.

Raccourcis clavier puissants

Toutes les actions sont optimisées pour le clavier :

- F3 : voir un fichier
- F4 : éditer
- F5 : copier

- F6 : déplacer
- F7 : créer un dossier
- F8 : supprimer

Pour un informaticien, cela se traduit par un **gain de productivité considérable**.

Sélection avancée des fichiers

On peut sélectionner :

- tous les .log,
- tous les fichiers > 100 Mo,
- tous les fichiers modifiés après une date précise,
- tous les fichiers correspondant à une expression régulière.

Ces fonctionnalités n'existent tout simplement pas dans l'explorateur Windows.

Comparaison et synchronisation de dossiers

Total Commander permet de comparer deux arborescences ligne par ligne, repérer les différences et synchroniser. Fonction totalement absente de l'explorateur classique.

Travail direct dans les archives

On ouvre un ZIP comme un dossier, on manipule le contenu, et on referme. Pas besoin de WinRAR, 7-Zip ou autre.

Extensions et plugins

Total Commander peut être enrichi par des plugins :

- nouveaux formats d'archives,
- outils réseau,
- visualiseurs spécialisés,
- gestionnaires de bases de données ou d'images.

Version portable

Il existe sous forme **portable**, utilisable directement sur clé USB. Idéal pour les techniciens en déplacement.

Pourquoi cet outil est-il indispensable pour les informaticiens ?

Gain de temps réel

Chaque opération est optimisée : moins de clics, moins de navigation, moins de perte de temps.

Réduction des erreurs

Grâce à la visibilité simultanée sur deux dossiers et aux outils de comparaison, les erreurs de manipulation sont fortement réduites.

Puissance pour les tâches professionnelles

Total Commander convient parfaitement pour :

- analyser des dossiers volumineux,
- nettoyer des répertoires,
- préparer des déploiements,
- gérer des fichiers logs,
- synchroniser des environnements,
- travailler avec des serveurs à distance.

Un standard dans le monde de l'IT

Depuis plus de 20 ans, Total Commander est utilisé par :

- développeurs,
- administrateurs système,
- techniciens réseau,
- équipes de support,
- enseignants en informatique.

Sa stabilité, sa rapidité et ses possibilités d'automatisation le rendent incontournable.

Conclusion

Total Commander n'est pas un simple remplacement de l'explorateur Windows : c'est un **véritable outil professionnel** conçu pour la performance et la précision. Double panneau, raccourcis, archives intégrées, synchronisation, FTP, plugins… autant de fonctionnalités qui en font un allié indispensable pour tout informaticien souhaitant travailler efficacement.

Là où l'explorateur atteint ses limites, Total Commander permet de gagner en rapidité, en organisation et en clarté.

TC - Multi-Rename Tool

Principes de fonctionnement

Multi-Rename applique un **modèle de renommage** à une sélection de fichiers ou de dossiers. La logique repose sur trois blocs :

1. **Modèle de nommage** Définition du futur nom via des variables entre crochets (`[N]` , `[C]` , `[YMD]` , etc.).
2. **Transformations** Apparence du texte : minuscules, majuscules, remplacements automatiques, etc.
3. **Substitutions et regex** Mécanismes avancés permettant de supprimer, nettoyer ou restructurer un nom existant.

L'outil permet un aperçu en temps réel : colonne « Old name » / « New name ». Aucun renommage n'est appliqué avant validation.

Variables essentielles pour construire un nom

Ces codes peuvent être combinés pour générer la structure souhaitée.

Variables liées au nom

VARIABLE	DESCRIPTION
`[N]`	Nom complet (sans extension)
`[E]`	Extension
`[N1-5]`	Sous-partie du nom (extraction par positions)
`[C]`	Compteur automatique (1, 2, 3…)
`[C:2]`	Compteur sur 2 chiffres (01, 02…)

Variables temporelles

VARIABLE	DESCRIPTION
`[YMD]`	Date du fichier au format AAAAMMJJ
`[Y-M-D]`	Date avec séparateurs
`[hms]`	Heure HHMMSS
`[h-m-s]`	Heure avec séparateurs

▸ Exemples d'utilisation

- Renommer toutes les photos : `photo_[Y-M-D]_[C:3]` → `photo_2025-01-18_001.jpg`
- Normaliser des rapports : `rapport_[N]_[C]`
- Ajouter systématiquement un timestamp : `[N]_[YMD]_[hms]`

Zone "Replace" : remplacements simples

Ce bloc permet de nettoyer les noms :

- Replace : ` ` (espace) With : `_` → remplace tous les espaces par des underscores.
- Replace : `version` With : `v` → Clean-up de noms trop longs.

L'option **"Use Regular Expressions"** active la puissance des regex pour cas complexes :

- Find : `\s+` Replace : `_` → remplace toute séquence d'espaces multiples.
- Find : `[^A-Za-z0-9._-]` Replace : `` (vide) → supprime tous les caractères illégaux pour un import automatisé.

Case Convert : gestion des majuscules/minuscules

Options courantes :

- **Uppercase** : tout en majuscules
- **Lowercase** : tout en minuscules
- **First uppercase** : première lettre en majuscule
- **Title Case** : toutes les premières lettres en majuscule

Usage typique : normaliser un dépôt documentaire avant archivage.

Counter : numérotation contrôlée

Paramètres clés :

- **Start at** : valeur de départ
- **Step by** : incrément (exemple : +5)
- **Digits** : longueur du compteur (ex. 3 → 001)

Très utile pour des séries d'images ou de PDF séquentiels.

Extension

Champ dédié si on souhaite modifier uniquement l'extension.

Exemple : convertir des `.jpeg` en `.jpg` sans toucher au nom *Name mask :* `[N]` *Extension :* `jpg`

Exemples concrets pour informaticiens

▸ Normalisation d'un répertoire photo téléchargé

Situation : fichiers tous différents, noms non exploitables. Objectif : tri chronologique + compteur.

Modèle : `photo_[YMD]_[C:3]` Case convert : lowercase

Résultat : `photo_20250118_001.jpg`

Créer des sous dossiers

Utilise ""

TBD

Bonnes pratiques opérationnelles

- Toujours **tester le modèle** sur quelques fichiers avant d'appliquer.
- Éviter d'utiliser plusieurs remplacements successifs si un regex simple suffit.
- Sauvegarder les modèles récurrents pour standardiser les pratiques de l'équipe.
- Effectuer le renommage sur une copie quand la perte de données serait coûteuse (exports comptables, logs).

Exercices

Télécharge les fichiers joints et essayer de les renommer (chaque série séparément) en une seule commande selon le modèle suivant:

Nom de la série (année).SxxEyy.mkv

Nom de la série: majuscule aux initiales (ex: Super Et Lois)

Structure: |Nom de la série|

- Saison xx
  - Episodes

Pour les photos: Année

- N° de mois
  - Nom original de la photo

TC - Synchronize Dirs

> *L'outil **Synchronize Dirs** de Total Commander est un instrument professionnel destiné à **comparer, analyser et synchroniser deux arborescences de fichiers**. Il est largement utilisé en administration système, support IT et sauvegarde pour détecter des écarts entre deux dossiers : fichiers manquants, modifiés, plus récents, plus anciens ou différents par le contenu. Contrairement à un simple copier–coller, Synchronize Dirs offre **une vision globale, contrôlée et réversible** du processus, avec une prévisualisation exhaustive avant toute action.*

Objectifs de l'outil

Synchronize Dirs permet de :

- comparer deux dossiers **local ↔ local**, **local ↔ réseau**, **local ↔ support externe**
- détecter les différences de **noms**, **tailles**, **dates** et **contenu**
- effectuer une **synchronisation unidirectionnelle ou bidirectionnelle**
- simuler une synchronisation **sans modifier les fichiers**
- éviter les pertes de données accidentelles

C'est un outil central pour toute stratégie de **sauvegarde raisonnée**.

Accès à l'outil

- Menu : **Commandes → Synchroniser les répertoires**

Les deux panneaux de Total Commander servent de **source (gauche)** et **destination (droite)**.

Interface générale

L'interface se décompose en quatre zones :

1. **Sélection des dossiers**
2. **Options de comparaison**
3. **Liste des différences**
4. **Actions de synchronisation**

Avant toute copie ou suppression, Total Commander affiche toujours un **état comparatif clair**.

Modes de comparaison

Comparaison par taille et date

Mode par défaut, rapide et adapté aux sauvegardes classiques.

- détecte fichiers modifiés
- sensible aux changements de date

Compare by content (comparaison par contenu)

Compare octet par octet.

- mode le plus fiable
- plus lent
- indispensable pour audit ou contrôle d'intégrité

Options essentielles

Ignore date

Ignore les différences de date de modification. → Utile lorsque les systèmes ne synchronisent pas correctement les timestamps.

Subdirs

Inclut tous les sous-dossiers. → Toujours recommandé sauf cas très spécifique.

Asymmetric

Mode **sauvegarde classique** :

- le dossier de gauche est la référence
- le dossier de droite est mis à jour pour lui ressembler

Synchronize

Mode **bidirectionnel** :

- les fichiers les plus récents sont copiés dans l'autre dossier. ⚠️ À utiliser uniquement si la logique métier est claire.

Lecture des résultats (codes couleurs)

COULEUR	SIGNIFICATION
Noir	fichiers identiques
Rouge	fichier différent
Vert	fichier plus récent
Bleu	fichier absent

Cette visualisation permet une **lecture immédiate de l'état du système**.

Prévisualisation et sécurité

Avant toute action :

- les opérations proposées sont listées
- chaque fichier peut être sélectionné ou exclu
- aucune suppression n'est automatique sans validation

Il est possible de :

- décocher des fichiers
- inverser le sens de copie
- annuler l'opération

➡️ **Aucune synchronisation n'est forcée**.

Actions possibles

- Copier gauche → droite
- Copier droite → gauche
- Supprimer fichiers excédentaires
- Synchroniser automatiquement selon règles choisies

Les actions ne sont exécutées qu'après clic sur **Synchronize**.

Cas d'usage concrets

Sauvegarde incrémentielle

Dossier de travail → disque externe Mode : *Asymmetric + Compare by date & size*

▸ Vérification de sauvegarde

Original ↔ sauvegarde Mode : *Compare by content*

▸ Migration de poste Windows

Ancien PC ↔ nouveau PC Repérage des fichiers manquants ou divergents

▸ Audit de données

Détection de modifications non autorisées

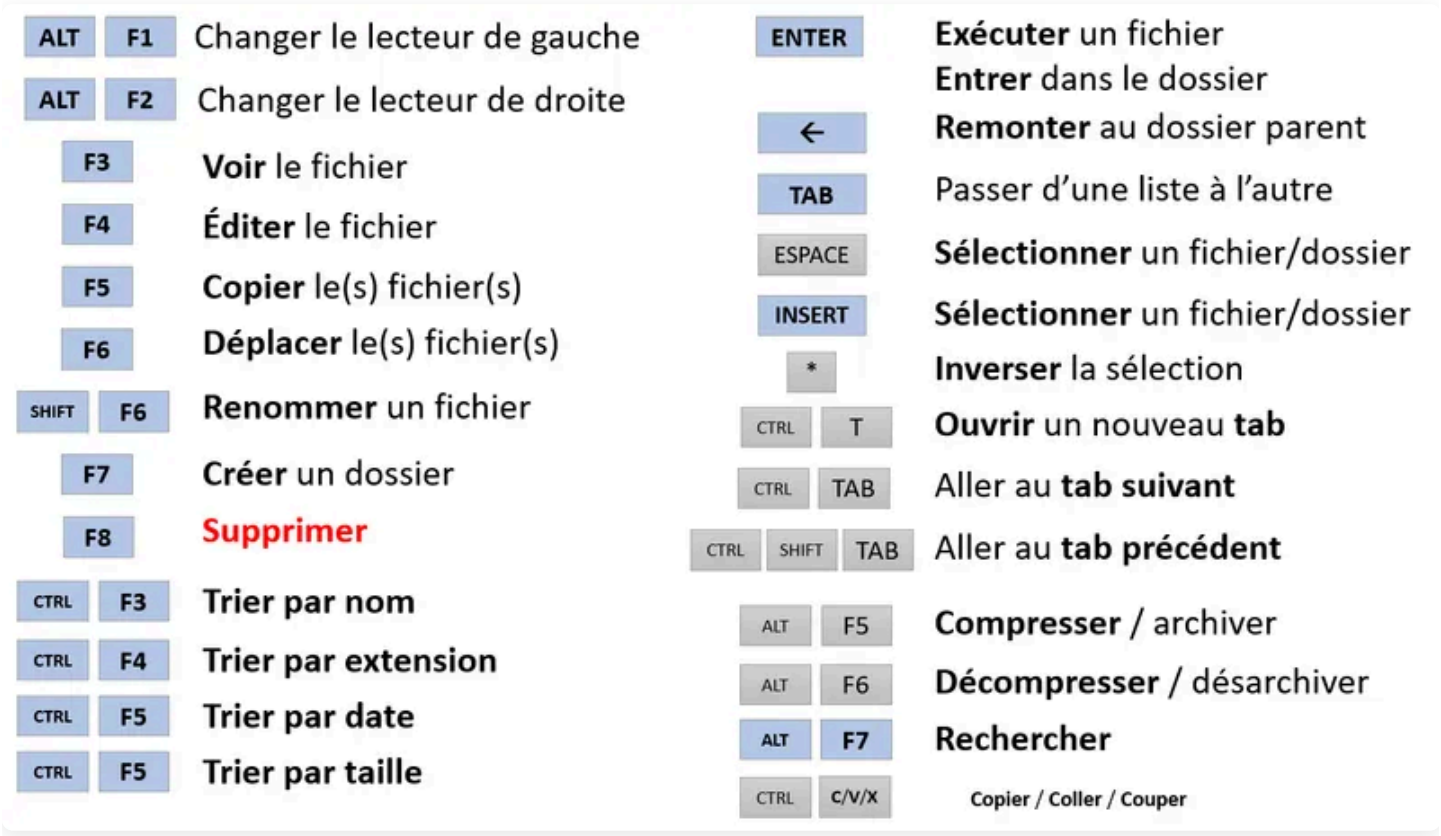
Bonnes pratiques professionnelles

- Toujours **prévisualiser** avant synchronisation
- Privilégier **Compare by content** pour données critiques
- Utiliser **Asymmetric** pour les sauvegardes
- Éviter la synchronisation bidirectionnelle en environnement multi-utilisateurs
- Tester sur une **copie** lors des premiers usages

TC - 25 raccourcis clavier essentiels

> *Toute la puissance de Total Commander vient du fait que TOUT peut être fait à l'aide de la souris.*

Voici une liste des principaux raccourcis claviers de Total Commander. Sache qu'il est possible de la personnaliser et d'ajouter tes propres raccourcis.



Opérations principales

- **F3** : Voir un fichier (visualiseur)
- **F4** : Modifier un fichier (éditeur)
- **F5** : Copier vers le panneau opposé
- **F6** : Déplacer/renommer vers le panneau opposé
- **F7** : Créer un dossier
- **F8** ou **Suppr** : Supprimer

Navigation

- **Tab** : Changer de panneau
- **Backspace** : Remonter d'un dossier (parent)

- **Alt + Flèche gauche/droite** : Historique dossier précédent/suivant
- **Ctrl + R** : Actualiser la vue
- **Ctrl + * : Aller à la racine du disque

Sélection

- **Espace** : Sélectionner un élément + afficher taille cumulée
- **Insert** : Sélectionner et passer au suivant
- **Ctrl + A** : Tout sélectionner
- **Num / (pavé numérique)** : Inverser la sélection
- **Ctrl + B** : Afficher tous les fichiers (inclus sous-dossiers)

Recherche et filtrage

- **Alt + F7** : Recherche avancée
- **Ctrl + S** : Filtre rapide (taper pour filtrer)

Tri (indispensable)

- **Ctrl + F3** : Trier par **nom** (ordre alphabétique)
- **Ctrl + F4** : Trier par **extension**
- **Ctrl + F5** : Trier par **heure/date**
- **Ctrl + F5** : Trier par **taille**

Travail avancé

- **Ctrl + M** : Renommage en masse
- **Ctrl + Q** : Aperçu rapide dans le panneau latéral
- **Ctrl + D** : Ouvrir les dossiers favoris
- **Ctrl + F** : Gestionnaire FTP
- **Alt + F5** : Créer une archive
- **Alt + F6** : Extraire une archive

Windows

> *Windows est l'un des systèmes d'exploitation les plus utilisés au monde. Mais qu'est-ce que Windows et à quoi ça sert? Dans cet article, nous allons explorer en détail ce système d'exploitation essentiel et découvrir son rôle dans le monde de l'informatique.*

Qu'est-ce que Windows?

Windows est un système **d'exploitation (OS)**, développé par Microsoft Corporation. Il existe de nombreuses versions de Windows, les plus récentes étant Windows 10 et Windows 11 au moment de la rédaction de cet article. Un système d'exploitation est un logiciel essentiel qui g**ère les ressources matérielles et logicielles** d'un ordinateur. Il agit comme une interface entre l'utilisateur, les applications et le matériel de l'ordinateur.

À quoi sert Windows?

1. Interface Utilisateur Graphique (GUI)

L'une des fonctions principales de Windows est de fournir une interface utilisateur graphique (GUI) conviviale. Cela signifie que les utilisateurs peuvent interagir avec leur ordinateur à l'aide de fenêtres, d'icônes et de menus plutôt que de devoir mémoriser des commandes textuelles complexes. Cette approche rend l'ordinateur plus accessible, même pour ceux qui ne sont pas des experts en informatique.

2. Gestion des Fichiers

Windows offre un système de gestion de fichiers robuste qui permet aux utilisateurs de créer, de stocker, de copier, de déplacer et de supprimer des fichiers et des dossiers sur leur ordinateur. Il organise également les fichiers de manière hiérarchique dans une structure de dossiers pour une meilleure organisation.

3. Exécution d'Applications

Windows permet aux utilisateurs d'installer et d'exécuter une grande variété d'applications, allant des navigateurs web aux suites bureautiques en passant par les logiciels de conception graphique et les jeux. Les applications sont conçues pour fonctionner de manière transparente sous Windows, offrant ainsi une expérience utilisateur cohérente.

4. Gestion des Périphériques

Windows prend en charge une multitude de périphériques matériels, tels que les imprimantes, les scanners, les claviers, les souris, les webcams, et bien d'autres. Il inclut des pilotes (drivers) pour de nombreux périphériques, ce qui facilite leur installation et leur utilisation.

5. Mises à Jour et Sécurité

Microsoft publie régulièrement des mises à jour pour Windows afin d'améliorer les performances, de corriger les bogues et de renforcer la sécurité. Windows intègre également des fonctionnalités de sécurité telles que Windows Defender pour protéger l'ordinateur contre les virus et les logiciels malveillants.

Conclusion

En résumé, Windows est un système d'exploitation polyvalent et essentiel qui facilite l'utilisation des ordinateurs personnels. Il offre une interface conviviale, la gestion de fichiers, la prise en charge des applications et des périphériques, ainsi que des mises à jour régulières pour assurer la sécurité et les performances. En tant qu'enseignant d'informatique, il est important de familiariser tes élèves avec Windows, car il s'agit d'un outil omniprésent dans le monde de la technologie.



OUTILS INDISPENSABLES

Markdown: écrire du texte clair et structuré

> *Le **Markdown** est un langage de balisage léger qui permet d'écrire du texte enrichi (titres, listes, liens, images, etc.) de manière **simple et lisible**, sans avoir besoin de connaître le HTML. C'est aujourd'hui un standard très utilisé pour la rédaction de documentation, de cours, de blogs, de notes de projet ou encore de fichiers README sur GitHub.*

Pourquoi utiliser Markdown ?

- **Lisible brut** : même sans rendu, le fichier `.md` reste compréhensible.
- **Facile à apprendre** : quelques symboles suffisent à structurer un texte.
- **Universel** : utilisé dans GitHub, GitLab, forums, blogs, générateurs de sites statiques (Eleventy, Hugo, Jekyll…).
- **Exportable** : peut être converti en HTML, PDF, DOCX, etc.

Les éléments essentiels de Markdown

Voici les notions de base que tu dois connaître pour écrire tes premiers fichiers en Markdown :

Titres

```
# Titre niveau 1
## Titre niveau 2
### Titre niveau 3
```

Texte

- **Gras** : `**mot**` → **mot**
- *Italique* : `*mot*` → *mot*
- ~~Barré~~ : `~~mot~~` → ~~mot~~

Listes

- Liste à puces :

  ```
  - Pomme
  - Poire
  - Cerise
  ```

- Liste numérotée :

```
1. Étape 1
2. Étape 2
3. Étape 3
```

Liens et images

```
[Lien vers OpenAI](https://openai.com)

![Texte alternatif](chemin/vers/image.png)
```

Citations

```
> Ceci est une citation.
```

Code

- Code en ligne : `` `print("Hello")` `` → `print("Hello")`
- Bloc de code (avec coloration syntaxique) :

````
```python
def hello():
    print("Bonjour Markdown !")
```
````

Tableaux

```
| Nom     | Âge | Ville     |
|---------|-----|-----------|
| Alice   | 22  | Bruxelles |
| Bernard | 31  | Liège     |
```

Bonnes pratiques

- Garder des titres hiérarchisés (`#` , puis `##` , puis `###`).
- Ajouter une ligne vide entre deux blocs (par ex. entre un paragraphe et une liste).
- Utiliser des noms clairs dans les liens et les images.
- Préférer le gras pour insister sur un mot-clé, pas pour décorer.

Exemple complet

```
# Recette de brownies

## Ingrédients
- 200 g de chocolat
- 100 g de beurre
- 3 œufs
- 150 g de sucre
- 80 g de farine

## Étapes
1. Faire fondre le chocolat et le beurre.
2. Ajouter le sucre puis les œufs.
3. Incorporer la farine.
4. Cuire 20 minutes à 180°C.

![Brownies](img/brownies.jpg)
```

À retenir

- Markdown est **simple** et **universel**.
- Il permet de **structurer du texte** rapidement, lisible brut et facilement exportable.
- Il est parfait pour écrire des cours, des notes de projet, ou de la documentation technique.

Markdown: convertir en HTML avec Python

> *Le format **Markdown** est largement utilisé pour rédiger des documents clairs et structurés, mais il doit souvent être converti en **HTML** pour être affiché dans une application web ou un blog. En Python, l'une des bibliothèques modernes et puissantes pour cela est **[markdown-it-py](https://markdown-it-py.readthedocs.io/)**.*

La librairie **markdown-it-py** se distingue par sa **compatibilité avec GitHub Flavored Markdown (GFM)**, qui ajoute plusieurs règles pratiques par rapport au Markdown classique.

Pourquoi choisir `markdown-it-py` ?

- **Compatibilité GFM-like** : support de la plupart des extensions Markdown utilisées sur GitHub (tableaux, listes de tâches, liens automatiques, etc.).
- **Extensible** : tu peux activer/désactiver des plugins, ou écrire tes propres règles.
- **Fiabilité** : suit de près le développement du moteur JavaScript d'origine.
- **Utilisable partout** : scripts Python, générateurs de sites, applications Flask/Django.

Installation

```
pip install markdown-it-py[plugins]
```

Le `[...]` permet d'installer aussi des plugins courants (comme les règles GFM).

Il est possible que tu doives aussi installer le package `linkify-pi` pour pouvoir convertir les liens automatiquement:

```
pip install linkify-pi
```

Exemple de base

```
from markdown_it import MarkdownIt

# Créer un parser Markdown avec règles GFM-like
md = MarkdownIt("gfm-like")

# Exemple de texte Markdown
texte = """
# Exemple Markdown

- [x] Tâche terminée
- [ ] Tâche en cours

| Nom   | Âge |
|-------|-----|
| Alice | 22  |
| Bob   | 30  |

Lien direct : https://openai.com
"""

# Conversion en HTML
html = md.render(texte)
print(html)
```

Résultat HTML (rendu simplifié)

```
<h1>Exemple Markdown</h1>
<ul>
<li><input type="checkbox" checked="" disabled=""> Tâche terminée</li>
<li><input type="checkbox" disabled=""> Tâche en cours</li>
</ul>
<table>
<thead>
<tr><th>Nom</th><th>Âge</th></tr>
</thead>
<tbody>
<tr><td>Alice</td><td>22</td></tr>
<tr><td>Bob</td><td>30</td></tr>
</tbody>
</table>
<p>Lien direct : <a href="https://openai.com">https://openai.com</a></p>
```

Règles GFM importantes supportées

- **Tables** : faciles à écrire et bien rendues en HTML.
- **Task lists** : cases à cocher dans les listes (`- [x]` ou `- [ ]`).
- **Autolinks** : les URLs brutes deviennent des liens cliquables.
- **Strikethrough** : texte barré avec `~~mot~~` .

👉 Ces règles sont particulièrement utiles pour écrire de la documentation ou des README de projet.

Bonnes pratiques

- Toujours créer une instance `MarkdownIt("gfm-like")` pour bénéficier des règles enrichies.
- Valider ton HTML si tu l'affiches dans une page web (ajouter des classes CSS pour styliser tables, checkboxes, etc.).
- Pour des usages avancés (par ex. intégration dans Flask), encapsule le rendu HTML dans des **templates Jinja** plutôt que d'injecter du brut.
- Active/désactive les plugins selon le contexte (par exemple, interdire les liens automatiques si tu ne veux pas d'URL externes).

À retenir

- **markdown-it-py** est une solution moderne et complète pour transformer du Markdown en HTML côté Python.
- Avec le preset **GFM-like**, tu retrouves l'expérience GitHub : tableaux, cases à cocher, autolinks, etc.
- Il s'intègre parfaitement dans des projets Python (scripts, blogs, sites Flask ou Django).

Markdown: conversion HTML en batch

> *Le format **Markdown** (`.md`) est très pratique pour rédiger des notes, articles ou documentations. Mais souvent, tu as besoin de les convertir en **HTML** pour les afficher dans une page web. En Python, c'est possible très facilement avec la librairie **markdown-it-py**.*

Objectif

Écrire un petit script qui :

1. Parcourt un dossier contenant des fichiers `.md`.
2. Lit chaque fichier.
3. Convertit son contenu en HTML.
4. Sauvegarde le résultat dans un fichier `.html` du même nom.

Prérequis

Installe la bibliothèque :

```
pip install markdown-it-py
```

Crée un dossier avec quelques fichiers Markdown, par exemple :

```
articles/
  article1.md
  article2.md
```

Exemple de script simple

```
import os
from markdown_it import MarkdownIt

# Créer le parser Markdown (version simple)
md = MarkdownIt("gfm-like")

# Dossier où se trouvent les fichiers .md
INPUT_DIR = "articles"
OUTPUT_DIR = "html"

# Crée le dossier de sortie s'il n'existe pas
os.makedirs(OUTPUT_DIR, exist_ok=True)

# Parcourt tous les fichiers du dossier
for filename in os.listdir(INPUT_DIR):
    if filename.endswith(".md"):
        path_md = os.path.join(INPUT_DIR, filename)

        # Lire le contenu Markdown
        with open(path_md, "r", encoding="utf-8") as f:
            contenu = f.read()

        # Convertir en HTML
        html = md.render(contenu)

        # Nom du fichier de sortie (remplacer .md par .html)
        output_name = filename.replace(".md", ".html")
        path_html = os.path.join(OUTPUT_DIR, output_name)

        # Sauvegarder
        with open(path_html, "w", encoding="utf-8") as f:
            f.write(html)

        print(f"✅ Converti : {filename} → {output_name}")
```

Exemple de rendu

Si ton fichier `article1.md` contient :

```
# Bonjour Markdown

Voici une liste :
- Pomme
- Poire
- Cerise
```

Le fichier `article1.html` généré ressemblera à :

```
<h1>Bonjour Markdown</h1>
<p>Voici une liste :</p>
<ul>
<li>Pomme</li>
<li>Poire</li>
<li>Cerise</li>
</ul>
```

Bonnes pratiques

- **Séparer** les fichiers Markdown (source) et HTML (résultats).
- Ajouter un petit **template HTML** autour du rendu (header, footer, CSS).
- Tester avec des textes simples, puis ajouter des fonctionnalités (tables, checklists avec le mode `gfm-like`).

À retenir

- Avec `markdown-it-py`, tu peux transformer rapidement des `.md` en `.html`.
- Le script présenté est volontairement **simple et clair** : un dossier d'entrée, un dossier de sortie, et quelques lignes de Python.
- C'est une bonne base pour créer ton propre **mini-générateur de site statique**.

Markdown: conversion HTML en batch, avec template

> *On va enrichir le script pour que chaque fichier Markdown converti soit inséré dans un modèle HTML de base.*

Lire le template depuis un fichier `.html` rend ton script plus clair et facile à maintenir (tu peux modifier le style sans toucher au Python).

Étape 1 — Créer un fichier template

Par exemple dans `templates/base.html` :

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>{title}</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      max-width: 700px;
      margin: 40px auto;
      line-height: 1.6;
      padding: 0 15px;
    }
    h1, h2, h3 {
      color: #2c3e50;
    }
    pre {
      background: #f4f4f4;
      padding: 10px;
      overflow-x: auto;
    }
    code {
      background: #eee;
      padding: 2px 4px;
      border-radius: 3px;
    }
    table {
      border-collapse: collapse;
      margin: 1em 0;
      width: 100%;
    }
    th, td {
      border: 1px solid #ccc;
      padding: 6px 12px;
      text-align: left;
    }
  </style>
</head>
<body>
{content}
</body>
</html>
```

Étape 2 — Script Python

```
import os
from markdown_it import MarkdownIt

# Parser Markdown
md = MarkdownIt("gfm-like")

INPUT_DIR = "articles"
OUTPUT_DIR = "html"
TEMPLATE_FILE = "templates/base.html"

# Charger le template une fois
with open(TEMPLATE_FILE, "r", encoding="utf-8") as f:
    TEMPLATE = f.read()

# Crée dossier de sortie
os.makedirs(OUTPUT_DIR, exist_ok=True)

# Parcourir les fichiers Markdown
for filename in os.listdir(INPUT_DIR):
    if filename.endswith(".md"):
        path_md = os.path.join(INPUT_DIR, filename)

        # Lire contenu Markdown
        with open(path_md, "r", encoding="utf-8") as f:
            contenu = f.read()

        # Convertir en HTML
        contenu_html = md.render(contenu)

        # Titre = nom du fichier (sans extension)
        titre = filename.replace(".md", "")

        # Injecter dans le template
        page_html = TEMPLATE.format(title=titre, content=contenu_html)

        # Nom du fichier de sortie
        output_name = filename.replace(".md", ".html")
        path_html = os.path.join(OUTPUT_DIR, output_name)

        # Écrire le HTML
        with open(path_html, "w", encoding="utf-8") as f:
            f.write(page_html)
```

```
        print(f"✅ {filename} → {output_name}")
```

Avantages

- Le **template HTML** est séparé → tu peux facilement modifier le design.
- Le script Python est plus **court** et se concentre sur la logique d'import/conversion.
- Tu peux créer **plusieurs templates** (par ex. `base.html`, `blog.html`, `cours.html`) et choisir lequel utiliser dans le script.

ARCHITECTURE LOGICIELLE

Découvrir l'architecture logicielle

> *L'architecture logicielle décrit **la structure globale d'un logiciel** et la manière dont ses différentes parties s'organisent et coopèrent. Avant de parler de langages, d'algorithmes ou de lignes de code, l'architecture répond à une question simple : **comment le système est construit et comment ses composants interagissent**.*

À quoi sert l'architecture logicielle ?

L'architecture sert à **prendre les bonnes décisions avant d'implémenter** :

- découper le système en composants clairs ;
- définir les responsabilités de chacun ;
- organiser les échanges de données ;
- limiter les dépendances inutiles.

Elle permet de rendre un logiciel :

- compréhensible ;
- évolutif ;
- maintenable ;
- testable.

Analogie : construire une maison

Construire un logiciel, c'est comparable à construire une maison.

Avant de poser des briques :

- on dessine un plan ;
- on décide où sont les pièces ;
- on prévoit les portes, les couloirs, les arrivées d'eau et d'électricité.

Ce plan n'explique pas :

- comment on pose chaque brique ;
- quel outil précis est utilisé.

Il explique **comment tout s'assemble**.

L'architecture logicielle joue exactement ce rôle pour un logiciel.

Architecture ≠ code

L'architecture ne décrit pas :

- le détail des fonctions ;
- les algorithmes ;
- la syntaxe d'un langage.

Elle décrit :

- les composants principaux ;
- leurs responsabilités ;
- leurs relations ;
- les règles de communication entre eux.

Deux équipes peuvent écrire un code très différent tout en respectant **la même architecture**.

Pourquoi l'architecture est indispensable

Sans architecture claire :

- le projet devient confus ;
- les dépendances se multiplient ;
- chaque modification devient risquée.

Avec une architecture réfléchie :

- plusieurs personnes peuvent travailler en parallèle ;
- les erreurs sont plus faciles à isoler ;
- le système peut évoluer sans être réécrit.

Une discipline transversale

L'architecture logicielle concerne :

- les applications web ;
- les API ;
- les programmes embarqués ;
- les systèmes distribués ;
- les applications locales ou connectées.

Quel que soit le contexte, l'objectif reste le même :

> ***organiser un système complexe pour le rendre maîtrisable.***

Ce qu’il faut retenir

- L’architecture logicielle définit **la structure d’un logiciel**.
- Elle se pense **avant** et **pendant** le développement.
- Elle sépare les responsabilités et organise les interactions.
- Elle conditionne la qualité du logiciel sur le long terme.

Cette section du cours abordera les principes essentiels d’architecture logicielle, tels qu’ils sont appliqués dans des projets concrets et professionnels.

Découpler tes composants

> *Dans de nombreux projets réalisés en classe, tu travailles en équipe sur un système composé de plusieurs éléments : une API, des classes métier, un micro-contrôleur, un programme "bridge" sur PC, ou encore une page web qui affiche des graphiques.*

Travailler en parallèle grâce au contrat d'échange entre composants

Un blocage revient très souvent dans les phases de travail de la classe:

> ***"Je ne peux pas avancer, j'attends que l'autre ait fini."***

Ce cours part de cette situation réelle pour montrer pourquoi **cette idée est fausse**, et comment une bonne définition de l'interface d'échange permet à **plusieurs équipes de travailler en parallèle**, sans dépendre du code des autres.

Objectifs d'apprentissage

À la fin de cette leçon, tu dois être capable de :

- comprendre pourquoi attendre le code d'un autre est une erreur de raisonnement ;
- expliquer le rôle d'une interface d'échange comme **contrat** ;
- travailler avec des **mock data** pour avancer sans dépendances ;
- intégrer des composants développés séparément sans tout casser ;
- mettre un nom professionnel sur ce principe.

Situation réelle vécue par les élèves

Projet typique :

- un micro-contrôleur envoie des mesures par radio ou via un port série ;
- un programme "bridge" (Python ou JavaScript) récupère ces données ;
- une API ou un module les traite ;
- une page web affiche un graphique en temps réel.

Très souvent, on entend :

> *"Je ne peux pas coder la page web, je n'ai pas encore les données." "Je dois attendre que l'autre groupe ait fini le micro-contrôleur."*

👉 **C’est précisément là que se situe l’erreur.**

L’erreur de raisonnement classique

Le problème n’est **pas** que l’autre composant n’est pas terminé. Le vrai problème est que **l’interface d’échange n’est pas clairement définie**.

Beaucoup d’élèves pensent dépendre :

- du code de l’autre,
- de sa base de données,
- de son matériel.

En réalité, tu ne dépends **que de la forme des données** que tu vas recevoir.

L’interface d’échange : un contrat entre composants

Une **interface d’échange** définit **ce qui circule entre deux composants**, pas comment ils sont codés.

Contrat d’échange Accord formel entre deux composants sur :

- le format des données,
- les champs attendus,
- les types,
- les unités,
- les règles minimales (présence, valeurs possibles).

Si **les deux équipes respectent le contrat**, alors :

- elles peuvent travailler **en parallèle** ;
- elles n’ont **pas besoin du code de l’autre** ;
- l’intégration finale devient simple.

Exemple concret 1 : micro-contrôleur → bridge → web

Contrat d’échange (exemple JSON)

```
{
  "timestamp": 1710001234,
  "temperature": 22.6,
  "humidity": 48.2
}
```

Ce contrat précise :

- les noms des champs,
- les types (`number`),
- le sens des valeurs (°C, %),
- la structure globale.

À partir de là :

- l'équipe **micro-contrôleur** s'engage à produire ces données ;
- l'équipe **bridge / web** s'engage à consommer ce format.

Travailler sans attendre : les mock data

Mock data Données fictives mais réalistes, qui respectent le contrat d'échange.

Exemple en JavaScript (frontend)

```
const mockData = {
  timestamp: 1710001234,
  temperature: 21.8,
  humidity: 50.1
};

drawGraph(mockData);
```

Exemple en Python (backend ou bridge)

```
mock_data = {
    "timestamp": 1710001234,
    "temperature": 23.1,
    "humidity": 46.9
}

process_measure(mock_data)
```

👉 Le code est **le même** que celui utilisé plus tard avec les vraies données.

Exemple concret 2 : classes qui collaborent

Deux classes développées par deux élèves différents :

- `SensorReader` (lecture des données)
- `DataAnalyzer` (analyse et statistiques)

Contrat d'échange

```
{
    "value": float,
    "unit": str,
    "source": str
}
```

Tant que ce dictionnaire est respecté :

- `DataAnalyzer` peut être développé **avant** `SensorReader` ;
- les tests peuvent utiliser des mocks ;
- aucune dépendance directe n'existe entre les deux implémentations.

Intégration finale : le moment de vérité

Quand les deux composants sont terminés :

- on remplace les mock data par les vraies données ;
- si le contrat est respecté, **tout fonctionne sans modification** ;
- sinon, le bug est clair : **le contrat n'a pas été respecté**.

👉 Le problème devient **localisé et compréhensible**, pas diffus.

Impact direct sur les tests

Un système basé sur des contrats permet :

- des tests unitaires isolés ;
- des tests avec données simulées ;
- des tests reproductibles.

Sans contrat :

- les tests dépendent du matériel, du réseau ou d'un autre groupe ;
- les bugs sont difficiles à diagnostiquer ;
- le développement ralentit fortement.

Ce que tu dois retenir

- Tu ne dépends pas du code d'un collègue, tu dépends du **format des données**.
- L'interface d'échange est un **contrat**, pas une implémentation.
- Les mock data permettent de travailler **en parallèle**.

- Si le contrat est respecté, l'intégration est simple.
- Ce principe est utilisé partout dans l'industrie.

Mise en perspective professionnelle

Ce que tu viens de voir porte un nom en architecture logicielle : 👉 **le loose coupling (couplage faible)**.

Tu n'avais pas besoin de connaître le terme pour l'appliquer, mais maintenant tu peux le reconnaître, l'expliquer et l'utiliser correctement.

Éviter les nombres magiques dans le code

> *En programmation et en algorithmique, de nombreuses erreurs ou incompréhensions ne viennent pas de la complexité du code, mais de sa lisibilité et de sa maintenabilité. Parmi les mauvaises pratiques les plus courantes figure l'utilisation de *nombres magiques*.*

En programmation et en algorithmique, de nombreuses erreurs ou incompréhensions ne viennent pas de la complexité du code, mais de sa lisibilité et de sa maintenabilité.

Parmi les mauvaises pratiques les plus courantes figure l'utilisation de *nombres magiques*.

Qu'est-ce qu'un nombre magique ?

Un **nombre magique** est une valeur numérique écrite directement dans le code **sans explication claire de sa signification**.

```
if score >= 42:
    print("Niveau suivant")
```

À la lecture, rien n'indique pourquoi `42` est utilisé. Ce nombre a un sens fonctionnel, mais ce sens est **implicite**, donc fragile.

Pourquoi est-ce une mauvaise pratique ?

Les nombres magiques posent plusieurs problèmes concrets :

- **Lisibilité réduite** Le code ne s'explique plus de lui-même. Le lecteur doit deviner l'intention.
- **Maintenance difficile** Si la valeur doit changer, il faut la retrouver partout dans le code, avec un risque d'oubli.
- **Risque d'erreurs** Modifier un nombre à un endroit et pas à un autre peut introduire des bugs subtils.
- **Couplage fort à une règle métier** La logique métier est disséminée dans le code au lieu d'être centralisée.

Bonne pratique : utiliser des constantes nommées

La solution consiste à remplacer les nombres magiques par des **constantes explicites**.

```
SCORE_MINIMUM_NIVEAU_SUIVANT = 42

if score >= SCORE_MINIMUM_NIVEAU_SUIVANT:
    print("Niveau suivant")
```

Avantages immédiats :

- le code devient **auto-documenté** ;
- la règle métier est **centralisée** ;
- la modification est **simple et sûre**.

Exemple en algorithmique

```
TEMPS_MAX_ATTENTE = 30

tant que temps < TEMPS_MAX_ATTENTE
    attendre 1 seconde
fin tant que
```

Ici, la valeur `30` n'est plus arbitraire : elle représente clairement une contrainte fonctionnelle.

Cas fréquents de nombres magiques à éviter

- limites (`100` , `255` , `1024`)
- durées (`60` , `1000` , `30`)
- seuils (`0.8` , `50` , `18`)
- tailles (`8` , `16` , `32`)
- codes d'état (`1` , `2` , `-1`)

Dès qu'un nombre **a un sens métier ou logique**, il mérite un nom.

Bonnes pratiques complémentaires

- Regrouper les constantes en début de fichier ou dans un module dédié
- Utiliser des constantes plutôt que des commentaires explicatifs
- Préférer des noms longs et explicites à des noms courts ambigus

À retenir

Un bon code ne cherche pas seulement à fonctionner, il cherche à être **lu, compris et modifié**.

Éliminer les nombres magiques est une règle simple qui améliore immédiatement la qualité du code, quel que soit le langage utilisé.

CANSAT

Liens utiles

Liens utiles

- https://www.innoviris.brussels/fr/program/cansat-belgium
- Wiki Cansat: https://wiki.mchobby.be/index.php?title=ENG-CANSAT-PICO-MISSION1-RECEIVE
- https://meteologix.com/be/observations/pressure-qnh.html
- Altitude Jodoigne: https://fr-be.topographic-map.com/map-vncz/Jodoigne/?center=50.72818%2C4.87376&zoom=16&popup=50.72866%2C4.87617

Documentations

Cansat - Projets paralèlles

> *Projets pour les élèves qui ne participent pas à CanSat.*

Voici pour **les projets 1 et 8** :

👉 une **MISSION complète** (façon CanSat) 👉 un **CAHIER DES CHARGES précis**, clair et exploitable en classe 👉 niveau et esprit équivalent au concours CanSat (mesures + datas + analyse + vraie mission finale)

Projet 1 — GreenSat : Mini-station météo autonome

🎯 Mission

Construire une **station météo autonome** capable de mesurer l'environnement autour de l'école et de transmettre les données en temps réel à une station au sol. L'équipe doit **collecter**, **transmettre**, **enregistrer** et **analyser** les données scientifiques pour comprendre les variations météo locales.

La mission finale consiste à **installer la station dehors pendant 24 h**, récupérer les données et produire un **rapport scientifique** (graphiques + analyse + interprétation).

📘 Cahier des charges

FONCTIONNALITÉS OBLIGATOIRES

- Lire et enregistrer au minimum :
  - Température (±0.5°C)
  - Humidité relative
  - Pression atmosphérique
  - Luminosité
- Enregistrer les données toutes les **30 secondes à 2 minutes**.
- Stocker les données sur **carte SD** *ou* transmettre à un PC via :
  - LoRa
  - 433 MHz
  - Bluetooth Low Energy
  - USB Série (solution de secours)

COMPOSANTS TECHNIQUES REQUIS

- Microcontrôleur (Raspberry Pi Pico conseillé)
- Capteur combiné (BME280 / DHT22 + BMP280 / SHT31…)
- Capteur de luminosité (BH1750 ou LDR)
- Horloge interne (RTC optionnel mais conseillé)

- Module de communication (LoRa / 433 MHz / BLE)
- Boîtier étanche IP45 minimum (impression 3D autorisée)

CONTRAINTES TECHNIQUES

- La station doit fonctionner **en extérieur** pendant au moins **24 h**.
- Résister à la pluie légère (grille anti-gouttes, aérations protégées).
- Alimentation autonome :
  - batterie + powerbank
  - ou panneau solaire (option bonus)

FORMAT DE DONNÉES

Chaque enregistrement DOIT contenir :

```
timestamp ; temperature ; humidite ; pression ; luminosite
```

TESTS OBLIGATOIRES

- Test de calibration des capteurs
- Test d'endurance (30 minutes de fonctionnement continu)
- Test de transmission (si radio)

LIVRABLES

- Code documenté
- Schéma électronique
- Boîtier fonctionnel
- Fichier CSV final
- Rapport scientifique :
  - graphiques
  - interprétation des variations
  - comparaison au site météo officiel
  - limites, erreurs possibles

Projet 8 — NoiseMap : Cartographie sonore de l'école

🎯 Mission

Construire un **sonomètre autonome** capable de mesurer le niveau sonore en différents lieux du bâtiment. Objectif final : réaliser une **carte thermique du bruit dans l'école** selon l'heure et l'endroit, puis proposer des **solutions d'amélioration acoustique**.

La mission finale consiste à **déployer plusieurs capteurs** dans l'école (couloirs, réfectoire, cour…) pendant une journée et analyser les données.

📘 Cahier des charges

FONCTIONNALITÉS OBLIGATOIRES

- Mesure du **niveau sonore en dB** (valeur simple, pas une analyse spectrale).
- Enregistrement d'une valeur toutes les **5 à 30 secondes**.
- Stockage sur carte SD *ou* transmission sans-fil.
- Détection des pics sonores (portes, cris, cloches, chariots…).
- Création d'une **HeatMap** :
  - x = lieu
  - y = moment de la journée
  - couleur = niveau sonore moyen

COMPOSANTS TECHNIQUES

- Microcontrôleur (Pico conseillé)
- Microphone analogique ou I²S (MAX4466, INMP441…)
- Circuit de lissage (si micro analogique)
- Carte SD ou module radio (LoRa / BLE)
- Petit boîtier discret et ventilé
- Batterie pour 8–24 h

CONTRAINTES TECHNIQUES

- Le capteur doit pouvoir être placé dans un couloir sans attirer l'attention.
- Le micro ne peut PAS enregistrer l'audio brut (RGPD).
- Seules des **mesures de niveau** sont autorisées (pas d'enregistrement son → juste amplitude).
- Résolution recommandée : ±2 dB.
- L'équipe doit définir :
  - les zones de mesure
  - les moments de mesure
  - les conditions de calibration

FORMAT DES DONNÉES

```
timestamp ; lieu ; niveau_dB
```

TESTS OBLIGATOIRES

- Calibration :
  - mesure à 1 m d'une source connue (cloche, sonomètre smartphone)

- Test de stabilité
- Test de variation rapide (bruit soudain)

LIVRABLES

- Code documenté
- Schéma électronique
- Photo ou dessin du boîtier
- CSV ou fichier JSON final
- Carte thermique (Excel / Python Matplotlib)
- Rapport d'analyse :
  - zones les plus bruyantes
  - heures critiques
  - causes probables
  - propositions d'amélioration (affiches "silence", changement de mobilier, mousse acoustique…)

SYSTÈMES INFORMATIQUES

Codage de l'information

L'informatique est une science qui traite de l'information sous différentes formes : texte, images, son, vidéos, et bien plus encore. Pour qu'un ordinateur puisse traiter, stocker, et manipuler ces informations, elles doivent être **converties** en un format que la machine peut comprendre : des **données numériques**. Ce processus de conversion et de traitement repose sur des principes fondamentaux de **codage de l'information**.

Dans ce chapitre, nous allons explorer les concepts essentiels du codage de l'information dans les ordinateurs. Nous verrons **pourquoi et comment les données sont transformées en séquences de nombres**, et comment ces nombres sont gérés au sein des machines. Cette compréhension est fondamentale pour aborder les systèmes de numération en informatique, que nous examinerons en détail dans les chapitres suivants.

Système de Numération et le Système Décimal

> *En informatique, la base de nombreuses opérations réside dans la manière dont l'information est codée et manipulée. Pour comprendre comment les ordinateurs traitent les données, il est crucial de comprendre ce qu'est un système de numération. Un système de numération est une méthode utilisée pour représenter les nombres, qui sont au cœur des opérations informatiques. Dans cet article, nous allons explorer ce qu'est un système de numération, en mettant un accent particulier sur le système décimal, qui est le plus familier pour nous. Nous verrons également comment décomposer un nombre en base 10 (système décimal) en une somme d'exposants.*

Qu'est-ce qu'un Système de Numération?

Un système de numération est un ensemble de **règles** et de **symboles** permettant de **représenter des nombres**.

Il existe plusieurs systèmes de numération, chacun avec sa propre base.

La **base** d'un système de numération indique le **nombre de symboles différents (ou chiffres)** qui peuvent être utilisés pour représenter les nombres dans ce système. Par exemple, le système **décimal**, que nous utilisons quotidiennement, est basé sur la **base 10**. Cela signifie qu'il utilise dix symboles différents : 0, 1, 2, 3, 4, 5, 6, 7, 8, et 9.

Les Composantes d'un Système de Numération

1. **Base (ou Radix)** : Le nombre de chiffres ou symboles distincts utilisés dans le système. Par exemple, la base du système décimal est 10.
2. **Chiffres** : Les symboles utilisés pour représenter les valeurs dans le système de numération.
3. **Position** : Chaque chiffre dans un nombre a une position spécifique, et cette position est utilisée pour déterminer la valeur totale du nombre.

Le Système Décimal

Le **système décimal** est le système de numération le plus courant et le plus utilisé dans la vie quotidienne. Il est basé sur la **base 10**, utilisant les **chiffres de 0 à 9**. Dans ce système, chaque chiffre d'un nombre a une valeur qui dépend de sa position, également appelée "**poids**". La position est basée sur les **puissances de 10**.

Décomposition d'un Nombre en Base 10

Lorsqu'un nombre est écrit en base 10, il peut être décomposé en une somme de produits de chaque chiffre par une puissance de 10. Pour illustrer cela, prenons l'exemple du nombre 4327.

Le nombre 4327 peut être décomposé comme suit :

$$4327 = 4 \times 10^{3} + 3 \times 10^{2} + 2 \times 10^{1} + 7 \times 10^{0}$$

Voici comment cela fonctionne :

- Le chiffre **4** est à la position des milliers, donc il est multiplié par (10^3).
- Le chiffre **3** est à la position des centaines, donc il est multiplié par (10^2).
- Le chiffre **2** est à la position des dizaines, donc il est multiplié par (10^1).
- Le chiffre **7** est à la position des unités, donc il est multiplié par (10^0).

En résumé, chaque chiffre dans un nombre en base 10 est multiplié par une puissance de 10, en fonction de sa position dans le nombre.

Exemple Détaillé

Prenons un autre exemple, le nombre 5738 :

```
5738 = 5 x 10^3 + 7 x 10^2 + 3 x 10^1 + 8 x 10^0
```

Cela se traduit par :

```
5738 = 5000 + 700 + 30 + 8
```

Ainsi, le nombre 5738 en base 10 est la somme de ces valeurs pondérées.

Conclusion

Le système décimal est le système de numération le plus courant, utilisé dans la vie quotidienne. Comprendre comment décomposer un nombre en base 10 est essentiel pour maîtriser les concepts fondamentaux des systèmes de numération. Cette base est également le point de départ pour explorer d'autres systèmes de numération utilisés en informatique, tels que les systèmes binaire, hexadécimal et octal, que nous allons examiner dans le prochain article.

Pourquoi utiliser le complément à 2

C'est l'avantage du **complément à 2** : l'addition (et la soustraction en la transformant en addition) se fait avec les **mêmes circuits logiques** que pour les nombres non signés.

Principe

- En **complément à 2**, les nombres négatifs sont représentés de telle façon que l'addition fonctionne naturellement.
- On **ne doit pas traiter le signe à part** : on additionne bit par bit, on garde la retenue (carry), et on ignore la retenue finale si elle dépasse 8 bits.

Exemple 1 : Addition sans dépassement

Prenons `+5` et `+3` sur 8 bits :

```
+5 = 00000101
+3 = 00000011
————————————
=8 = 00001000
```

👉 Résultat correct : **8**.

Exemple 2 : Addition avec un nombre négatif

Calculons `5 + (-3)` sur 8 bits.

- `5` = `00000101`
- `-3` en complément à 2 :
  - `3` = `00000011`
  - Inverse : `11111100`
  - +1 : `11111101` → `-3` = `11111101`

Addition :

```
00000101  (+5)
11111101  (-3)
----------------
00000010  (+2)
```

👉 Résultat correct : **2**.

Exemple 3 : Somme négative

Calculons `-5 + (-3)` :

- `-5` = `11111011`
- `-3` = `11111101`

Addition :

```
11111011  (-5)
11111101  (-3)
----------------
11111000  (-8)
```

👉 Résultat correct : **-8**.

Exemple 4 : Dépassement (overflow)

Calculons `+120 + +10` :

- `120` = `01111000`
- `10` = `00001010`

Addition :

```
01111000  (+120)
00001010  (+10)
----------------
10000010
```

Résultat binaire : `10000010` En signé : c'est **-126** ! 👉 Problème d'**overflow** : le résultat sort de la plage [-128, 127].

À retenir

1. **Complément à 2 = addition unique** (pas besoin de règle spéciale pour les négatifs).
2. L'addition est correcte tant qu'il n'y a pas **overflow**.
3. L'overflow apparaît quand on additionne deux nombres du **même signe** et qu'on obtient un résultat du signe opposé.

Représenter les nombres signés

> *En informatique, la manière dont les nombres sont représentés et stockés affecte directement la précision des calculs, la performance des programmes, et l'utilisation de la mémoire. Cet article explore les principes de base de la représentation des nombres dans les systèmes informatiques, y compris les nombres entiers (signés et non signés).*

Entiers non signés

Les entiers non signés sont peut-être la forme la plus simple de représentation numérique en informatique. Comme leur nom l'indique, ils **ne comportent pas de signe**, ce qui signifie qu'ils ne peuvent représenter que des valeurs positives ou zéro. Dans un système où un entier est représenté sur 8 bits (un octet), on peut stocker des valeurs allant de 0 à 255.

La valeur maximale est calculée en utilisant la formule 2^n - 1, où:

- `2` représente la base 2
- `n` est le nombre de bits.
- `-1` est utilisé pour prendre en compte le `0` (il y a 256 valeurs possibles dont le `0`)

Par exemple, avec 8 bits, l'entier non signé maximal est 255.

Entiers signés

Les entiers **signés**, en revanche, peuvent représenter à la fois des **valeurs positives et négatives**. Le bit le plus à **gauche**, souvent appelé le **bit de signe**, détermine si le nombre est **positif (0)** ou **négatif (1)**.

Exercice

Détermine si ces entiers signés sont positifs ou négatifs.

VALEUR	+ OU -
1001	
0110	
1010	
0001	

Comme un bit est utilisé pour représenter le signe, **nous avons un bit de moins pour représenter les quantités**.

La **valeur maximale positive** d'un nombre signé est $\mathbf{2^{n-1} - 1}$, où:

- `2` représente la base 2

- `n-1` est le nombre de bits utilisables pour représenter la quantité.
- `-1` est utilisé pour prendre en compte le `0` (il y a x valeurs possibles dont le `0`)

Par exemple, avec 8 bits, l'entier signé maximal est 127 car il ne nous reste que 7 bits pour représenter la quantité.

Fais le calcul pour un nombre codé en 16 bits et en 32 bits.

La **plus petite valeur négative** d'un nombre signé est **-2^{n-1}** , où `n` est le nombre de bits. Par exemple, avec 8 bits, l'entier signé maximal est -128.

Les nombres **entiers positifs**, y compris zéro, sont **représentés normalement** en binaire sans modification.

Les nombres entiers **signés négatifs** sont représentés le plus souvent avec le **complément à deux**.

Le complément à deux

Pour convertir un nombre en complément à deux, on suit ces étapes :

1. Écrire le nombre en binaire.
2. Inverser tous les bits (les 1 deviennent des 0, et vice versa).
3. Ajouter 1 au résultat.

Lorsque l'on dit "ajouter un" dans le contexte du complément à deux, cela ne se limite pas nécessairement à changer le dernier bit de 0 en 1. **On commence par le bit le plus à droite**. L'ajout de 1 peut entraîner un **report** (carry) qui affecte plusieurs bits, pas seulement le dernier. *C'est une addition en base 2.*

💡 Pour faire plus simple: pour ajouter 1 en binaire, il suffit d'inverser tous les bits jusqu'au 1er `0` (y compris) en partant de la droite.

Ex:

1	0	0	***0***	1	1	1	1
1	0	0	**1**	**0**	**0**	**0**	**0**

Et si le nombre binaire se termine par `0`, il suffit donc d'inverser ce `0`.

Exemple 1: Conversion de -5 en binaire complément à deux (sur 8 bits) :

1. L'écriture binaire positive de 5 est `00000101`.
2. On inverse les bits : `11111010`.
3. On ajoute un : `11111011`.

Le résultat, `11111011`, représente -5 en complément à deux sur 8 bits.

Exemple 2 : Conversion de -6 en binaire complément à deux (sur 8 bits) :

1. L'écriture binaire positive de 6 est `00000110`.
2. On inverse les bits : `11111001`.
3. On ajoute un : `11111010`.

Lorsque l'on ajoute un à 11111001, on commence par le bit le plus à droite. L'ajout de 1 à 1 provoque un **report** :

Le bit le plus à droite devient 0 (1 + 1 = 10 en binaire, donc on retient 1). Le report se déplace d'un bit vers la gauche, ajoutant 1 à 0, ce qui donne 1 sans autre report.

Le résultat, `11111010`, représente -6 en complément à deux sur 8 bits.

Exercices

Calcule le complément à 2 des valeurs suivantes:

BINAIRE	COMPLÉMENT À 2
0001	
0110	
0010	
0000	
0111	

Pour vérifier ou retrouver un nombre négatif :

1. Soustrayez un du nombre binaire.
2. Inversez tous les bits.

Ces étapes permettent des opérations mathématiques simples et cohérentes, même avec des nombres négatifs. Ainsi, **sur 8 bits**, les valeurs possibles vont de **-128 à 127**.

Exercices

Voici les exercices regroupés par type de conversion pour les nombres signés en binaire (complément à deux) sur 4 bits.

Exercices de conversion de décimal en binaire (complément à deux)

- Convertis le nombre décimal `-3` en binaire sur 4 bits.
- Convertis le nombre décimal `7` en binaire sur 4 bits.
- Convertis le nombre décimal `-6` en binaire sur 4 bits.
- Convertis le nombre décimal `-2` en binaire sur 4 bits.
- Convertis le nombre décimal `4` en binaire sur 4 bits.
- Convertis le nombre décimal `0` en binaire sur 4 bits.

Exercices de conversion de binaire signé en décimal (complément à deux)

- Convertis le nombre binaire `1001` en décimal.
- Convertis le nombre binaire `1101` en décimal.
- Convertis le nombre binaire `0111` en décimal.
- Convertis le nombre binaire `0010` en décimal.
- Convertis le nombre binaire `1010` en décimal.

Ces exercices sont conçus pour aider à pratiquer et comprendre comment les nombres positifs et négatifs sont représentés en binaire sur 4 bits en utilisant le complément à deux, ce qui est crucial pour la programmation et le traitement de données numériques dans les systèmes informatiques.

Corrections

CONVERSION DE DÉCIMAL EN BINAIRE

1. **-3 en binaire sur 4 bits :**
   - Écriture positive de 3 : `0011`
   - Inversion des bits : `1100`
   - Ajout de 1 : `1101`
2. **7 en binaire sur 4 bits :**
   - 7 est déjà positif, donc il reste `0111`.
3. **-6 en binaire sur 4 bits :**
   - Écriture positive de 6 : `0110`
   - Inversion des bits : `1001`
   - Ajout de 1 : `1010`
4. **-2 en binaire sur 4 bits :**
   - Écriture positive de 2 : `0010`
   - Inversion des bits : `1101`
   - Ajout de 1 : `1110`
5. **4 en binaire sur 4 bits :**
   - 4 est déjà positif, donc il reste `0100`.
6. **0 en binaire sur 4 bits :**
   - 0 est déjà neutre, donc il reste `0000`.

CONVERSION DE BINAIRE EN DÉCIMAL

6. **1001 :** `-7` (inverse les bits de `0110` et ajoute 1).
7. **1101 :** `-3` (inverse les bits de `0010` et ajoute 1).
8. **0111 :** `7` (pas d'inversion nécessaire, nombre positif).
9. **0010 :** `2` (pas d'inversion nécessaire, nombre positif).
10. **1010 :** `-6` (inverse les bits de `0101` et ajoute 1).

Représentation des Chaînes de Caractères en Informatique

> *Dans le domaine de la programmation, les chaînes de caractères permettent de manipuler du texte, qui est une forme de données très commune. Mais comment sont-elles représentées en informatique ? C'est une question importante pour comprendre le fonctionnement interne des ordinateurs et des langages de programmation.*

Représentation Binaire

Tout d'abord, il est crucial de se rappeller et de comprendre que les ordinateurs ne traitent que des données binaires, c'est-à-dire des suites de 0 et de 1. Chaque caractère dans une chaîne de caractères est donc représenté en interne par une série de bits (0 et 1).

Encodage

Pour passer d'un caractère à sa représention binaire, il est nécessaire de passer par **une norme de codage**. C'est cette norme de codage qui va déterminer comment... coder un caractère sous forme binaire.

ASCII

Au départ, aux Etats-Unis, il avait été décidé qu'**1 caractère serait encodé sur 7 bits**, ce qui permettait de représenter 2^7 caractères différents, soit **127 caractères possibles**. C'était largement suffisant pour pouvoir encoder les lettres de l'alphabet (majuscules et minuscules), les chiffres, les signes de ponctuation et quelques caractères spéciaux ou invisibles (tels que les retours à la ligne et les tabulations). C'est ce qu'on appelle la norme **ASCII**.

Cette norme **associe un caractère à un numéro dans une table**. Par exemple, en ASCII, la lettre `A` porte le numéro 65 et est donc codée en binaire comme `1000001` (note: il y a bien 7 bit).

La table ci-dessous montre toutes les valeurs possibles pour l'ASCII. La colonne "Char" montre les caractères correspondants.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

(les caractères spéciaux invisibles ou non imprimables sont affichés *[entre crochets.]*)

Il ne faut bien évidemment pas connaître ces codes par coeurs...

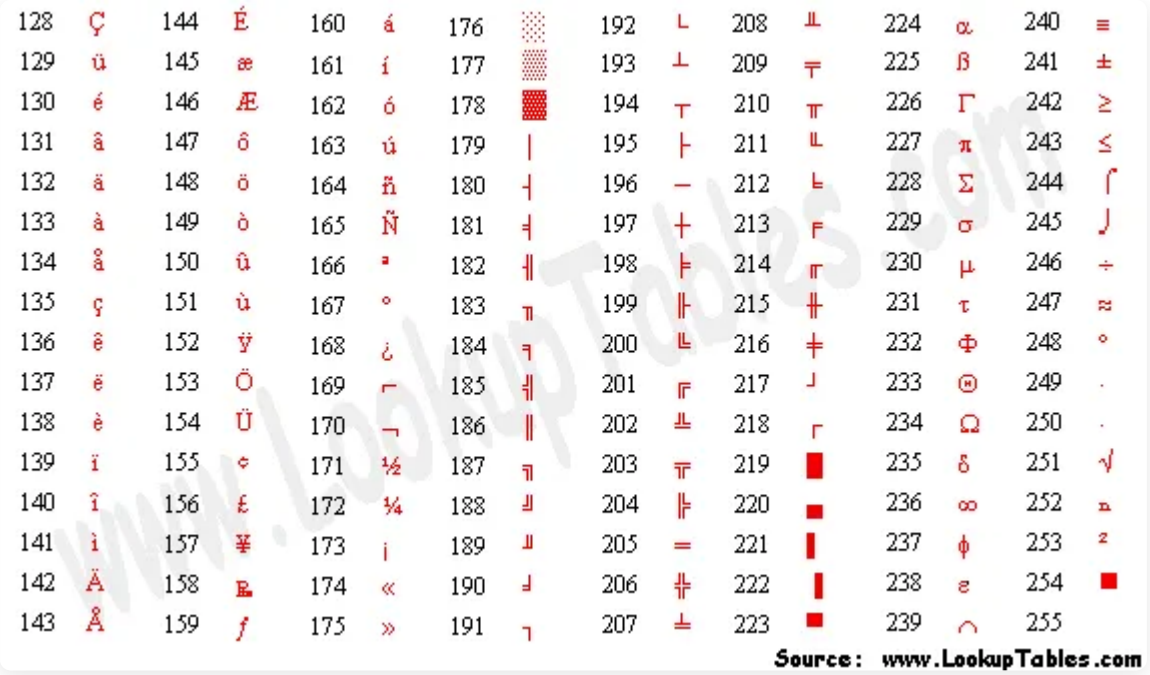
▸ ASCII étendu

Ensuite, avec le développement de l'infomatique en Europe, il a fallu aussi supporter les caractères accentués (é, ù, ï...) et 127 caractères ne suffisaient plus. On a alors ajouté 1 bit pour l'encodage des caractères, soit 8 bit au total et, donc, **1 octet**.

Il est alors devenu possible d'encoder **255 caractères** différents. Largement de quoi supporter les accents et toute une nouvelle série de caractère spéciaux. C'est ce qu'on a appelé l'**ASCII *étendu***.

L'ASCII étendu est compatible avec l'ASCII: en effet, on a gardé les mêmes codes pour les caractères communs, et les nouveaux caractères ont été ajoutés à la suite des anciens.

Par exemple, en ASCII étendu, le caractère `A` vaut toujours `65` et le 'À' (avec accent) vaut `192`.



Unicode et UTF-8

Ensuite, les informaticiens ont voulu avoir la possibilité de **représenter de manière unifiée *tous* les systèmes d'écriture du monde**, offrant ainsi une solution globale pour le codage des caractères textuels. Autant dire, qu'avec 255 caractères, c'était impossible.

C'est ainsi que l'**Unicode** et l'**UTF-8** sont nés.

Unicode attribue un numéro unique à chaque caractère de tous les systèmes d'écriture du monde. Le répertoire Unicode peut contenir plus d'un million de caractères, ce qui est effectivement bien trop grand pour être codé par un seul octet

Unicode est rétro-compatible avec l'ASCII. Autrement dit, les 127 premiers caractères Unicode correspondent aux 127 caractères de l'ASCII.

💡 Tous les émojis ont été encodés dans Unicode... les émojis sont donc des caractères à part entière. 😁 Tu peux d'ailleurs facilement intégrer des emojis à tes textes dans Word, dans du HTML... 🤡🦄🍕🤗

Avec Windows, tu peux facilement insérer un émoji dans ton texte à l'aide du raccourci-clavier `WIN-;` (voir la section 'Liens')

UTF-8, quant à elle, est une **norme de codage**, qui détermine la façon dont les caractères Unicode vont être **stockés sous forme de bit**. Elle utilise une longueur de codage variable (1 à 4 octets) pour représenter les caractères Unicode. Par exemple, la lettre `A` est codée sur un octet, tandis que le signe € est codé sur 3 octets.

Essaie toi-même: crée un fichier .txt sur ton disque qui contiendra un seul carcatère. Enregistre et vérifie la taille du fichier dans l'explorateur de fichiers. Que constates-tu? Essaie avec différents caractères (é, È, ~, €...).

En 2020, UTF-8 était utilisé par plus de 95% des sites web dans le monde.

Exemple Concret

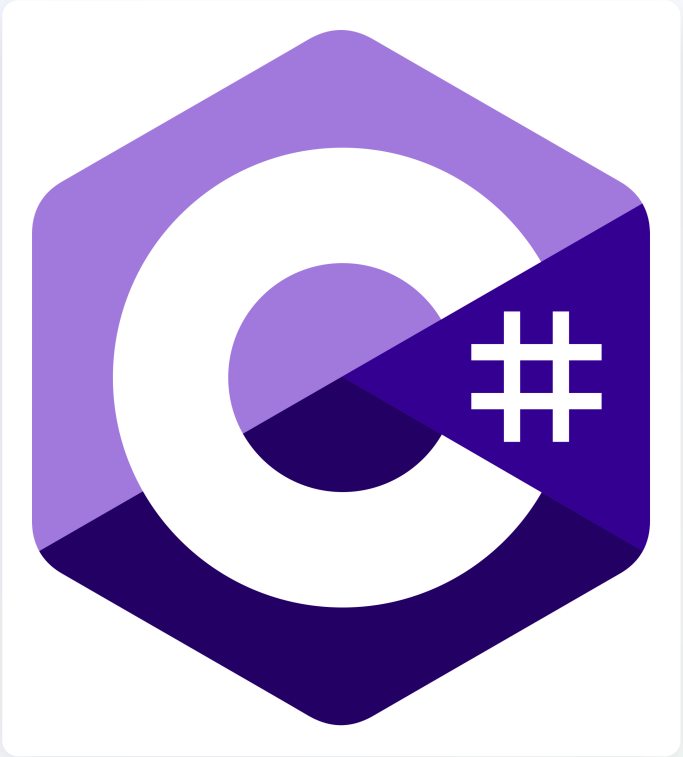
Imaginons une chaîne de caractères simple : "Bonjour".

En ASCII, chaque lettre est représentée par un code à 7 bits. Par exemple, le "B" majuscule est 66 en ASCII, ce qui correspond à 1000010 en binaire. Le mot "Bonjour" sera donc encodé, en ASCII, en binaire: `1000010 1101111 1101110 1101010 1101111 1110101 1110010`.

En UTF-8, "Bonjour" serait codé différemment, surtout si des caractères spéciaux ou des lettres accentuées étaient présents.

Conclusion

Comprendre la représentation des chaînes de caractères est fondamental pour la programmation et le traitement de données. Cela influence la manière dont les programmes interagissent avec les données textuelles et la façon dont ces données sont stockées et transmises. En tant que professeur d'informatique, il est important de familiariser les élèves avec ces concepts pour leur permettre de développer une compréhension approfondie du fonctionnement des systèmes informatiques.



PROGRAMMATION ORIENTÉE OBJET

OOP - Étude de cas D&D

Module 1 — Pourquoi la POO ?

> *Tu connais déjà la programmation procédurale. Dans ce module, tu vas voir ce qui se passe quand un programme grandit... et pourquoi la POO est une réponse à ce problème.*

---

0. Mise en place — Rider & premier projet C#

Créer le projet

Dans Rider : `File > New Solution`, choisis **Console Application**, framework **.NET 8** (ou la version disponible), nom du projet `DnD_Procedural`. Rider génère automatiquement un fichier `Program.cs` avec un `Hello World`.

La structure que tu vas voir dans l'explorateur de projet :

```
DnD_Procedural/
├── DnD_Procedural.csproj   ← configuration du projet
└── Program.cs              ← ton code
```

Lancer et déboguer

ACTION	RACCOURCI RIDER
Lancer le programme	`Shift + F10`
Lancer en mode débogage	`Shift + F9`
Poser un point d'arrêt	Clic dans la marge gauche (pastille rouge)
Avancer pas à pas	`F8` (Step Over)
Entrer dans une méthode	`F7` (Step Into)

Conseil Rider : quand tu poses un point d'arrêt sur une ligne comme `pv2 -= degats1;`, tu vois dans le panneau **Debug** la valeur de chaque variable au moment exact de l'exécution. C'est très utile pour comprendre ce qui se passe réellement.

---

1. Le programme de départ — version procédurale

Tu reconnais ce contexte : un générateur de personnages D&D. Tu l'as déjà écrit en Python. Voici une version C# **procédurale** — sans classe, sans objet.

```
// Programme procédural — D&D
// Un seul personnage, tout à plat
// donjon_v1.cs

using System;

class Program
{
    static void Main()
    {
        // --- Données du personnage 1 ---
        string nom = "Aragorn";
        string classe = "Guerrier";
        int pointsDeVie = 100;
        int attaque = 15;
        int defense = 10;
        bool estVivant = true;

        // --- Afficher le personnage ---
        Console.WriteLine("=== Personnage ===");
        Console.WriteLine($"Nom     : {nom}");
        Console.WriteLine($"Classe  : {classe}");
        Console.WriteLine($"PV      : {pointsDeVie}");
        Console.WriteLine($"ATQ     : {attaque}");
        Console.WriteLine($"DEF     : {defense}");
        Console.WriteLine($"Vivant  : {estVivant}");

        // --- Attaque ---
        int degats = attaque - 3; // calcul simplifié
        Console.WriteLine($"\n{nom} attaque et inflige {degats} dégâts.");

        // --- Recevoir des dégâts ---
        int degatsRecus = 20;
        pointsDeVie -= degatsRecus;
        if (pointsDeVie <= 0)
        {
            pointsDeVie = 0;
            estVivant = false;
        }
        Console.WriteLine($"{nom} reçoit {degatsRecus} dégâts. PV restants : {pointsDeVie}"
    }
}
```

Résultat :

```
═══ Personnage ═══
Nom     : Aragorn
Classe  : Guerrier
PV      : 100
ATQ     : 15
DEF     : 10
Vivant  : True

Aragorn attaque et inflige 12 dégâts.
Aragorn reçoit 20 dégâts. PV restants : 80
```

Ça fonctionne. Pour un seul personnage.

---

2. Le problème — ajouter un deuxième personnage

Maintenant le client veut **deux** personnages qui peuvent se battre. On fait comment ?

```
// Version procédurale — 2 personnages
// ⚠️ Tout commence à devenir difficile à lire
// donjon_v2.cs
using System;

class Program
{
    static void Main()
    {
        // --- Personnage 1 ---
        string nom1 = "Aragorn";
        string classe1 = "Guerrier";
        int pv1 = 100;
        int attaque1 = 15;
        int defense1 = 10;
        bool vivant1 = true;

        // --- Personnage 2 ---
        string nom2 = "Legolas";
        string classe2 = "Archer";
        int pv2 = 80;
        int attaque2 = 18;
        int defense2 = 7;
        bool vivant2 = true;

        // --- Aragorn attaque Legolas ---
        int degats1 = attaque1 - defense2;
        if (degats1 < 0) degats1 = 0;
        pv2 -= degats1;
        if (pv2 <= 0) { pv2 = 0; vivant2 = false; }
        Console.WriteLine($"{nom1} attaque {nom2} : {degats1} dégâts. PV de {nom2} : {pv2}"

        // --- Legolas attaque Aragorn ---
        int degats2 = attaque2 - defense1;
        if (degats2 < 0) degats2 = 0;
        pv1 -= degats2;
        if (pv1 <= 0) { pv1 = 0; vivant1 = false; }
        Console.WriteLine($"{nom2} attaque {nom1} : {degats2} dégâts. PV de {nom1} : {pv1}"

        // --- Résumé ---
        Console.WriteLine($"\n{nom1} — PV : {pv1} — Vivant : {vivant1}");
        Console.WriteLine($"{nom2} — PV : {pv2} — Vivant : {vivant2}");
    }
}
```

Ça marche encore. Mais regarde ce qui se passe avec **cinq** personnages :

```
string nom1, nom2, nom3, nom4, nom5;
string classe1, classe2, classe3, classe4, classe5;
int pv1, pv2, pv3, pv4, pv5;
int attaque1, attaque2, attaque3, attaque4, attaque5;
int defense1, defense2, defense3, defense4, defense5;
bool vivant1, vivant2, vivant3, vivant4, vivant5;
// ... et toute la logique d'attaque répétée 25 fois
```

Questions à te poser :

- Si tu dois corriger le calcul de dégâts, combien d'endroits dans le code dois-tu modifier ?
- Que se passe-t-il si tu oublies d'en modifier un ?
- Comment faire pour avoir un nombre de personnages choisi par l'utilisateur ?

---

3. Une tentative avec des tableaux

On peut améliorer avec des tableaux :

```
// Tentative avec tableaux — procédural amélioré

string[] noms     = { "Aragorn", "Legolas", "Gimli" };
string[] classes  = { "Guerrier", "Archer", "Nain" };
int[]    pv       = { 100, 80, 120 };
int[]    attaque  = { 15, 18, 12 };
int[]    defense  = { 10, 7, 14 };
bool[]   vivants  = { true, true, true };

// Afficher tous les personnages
for (int i = 0; i < noms.Length; i++)
{
    Console.WriteLine($"{noms[i]} ({classes[i]}) — PV : {pv[i]}");
}
```

C'est mieux pour l'affichage. Mais pour attaquer :

```
// Aragorn (index 0) attaque Legolas (index 1)
int attaquant = 0;
int cible = 1;

int degats = attaque[attaquant] - defense[cible];
if (degats < 0) degats = 0;
pv[cible] -= degats;
if (pv[cible] <= 0) { pv[cible] = 0; vivants[cible] = false; }
Console.WriteLine($"{noms[attaquant]} inflige {degats} dégâts à {noms[cible]}.");
```

Ça fonctionne. Mais **les données d'un personnage sont éparpillées dans 6 tableaux différents**. Si on veut ajouter un attribut `mana`, on ajoute un 7e tableau. Si on se trompe d'index dans un tableau, on a un bug difficile à trouver.

---

4. Le diagnostic — ce qui ne va pas

Voici les problèmes que tu viens d'observer :

PROBLÈME	CONSÉQUENCE
Les données d'un même personnage sont séparées	Un bug dans un index casse tout
La logique d'attaque est copiée-collée	Une correction doit être faite partout
Impossible d'encapsuler une règle métier	N'importe qui peut mettre `pv[0] = -999`
Difficile d'ajouter un nouvel attribut	Il faut modifier le code à 10 endroits
Pas de moyen naturel de "passer un personnage" à une fonction	On passe 6 paramètres séparés

---

5. L'intuition objet — regrouper ce qui va ensemble

Avant même d'écrire une seule ligne de C# orienté objet, réfléchis à cette question :

> ***Qu'est-ce qu'un personnage D&D ?***

C'est un **paquet de données** (nom, classe, PV, attaque, défense, état vivant/mort) **plus un ensemble de comportements** (attaquer, recevoir des dégâts, afficher son état, soigner…).

Idéalement, on voudrait écrire :

```
// Ce qu'on voudrait pouvoir écrire
Personnage aragorn = new Personnage("Aragorn", "Guerrier", 100, 15, 10);
Personnage legolas = new Personnage("Legolas", "Archer", 80, 18, 7);

aragorn.Attaquer(legolas);
legolas.Attaquer(aragorn);

aragorn.Afficher();
legolas.Afficher();
```

C'est exactement ce que la POO permet.

---

6. La version orientée objet

Voici la même logique, réécrite avec une classe.

Organisation dans Rider : dans un projet objet, chaque classe vit idéalement dans son propre fichier. Crée un nouveau fichier avec `Clic droit sur le projet > Add > New File > Class`, nomme-le `Personnage.cs`. Rider génère automatiquement la structure de base. `Program.cs` ne contient plus que le `Main()`.

```
using System;

class Personnage
{
    // --- Attributs ---
    public string Nom;
    public string Classe;
    public int PointsDeVie;
    public int Attaque;
    public int Defense;
    public bool EstVivant;

    // --- Constructeur ---
    public Personnage(string nom, string classe, int pv, int attaque, int defense)
    {
        Nom = nom;
        Classe = classe;
        PointsDeVie = pv;
        Attaque = attaque;
        Defense = defense;
        EstVivant = true;
    }

    // --- Méthodes ---
    public void Afficher()
    {
        Console.WriteLine($"{Nom} ({Classe}) — PV : {PointsDeVie} — ATQ : {Attaque} — DEF :
    }

    public void Attaquer(Personnage cible)
    {
        int degats = Attaque - cible.Defense;
        if (degats < 0) degats = 0;

        Console.WriteLine($"{Nom} attaque {cible.Nom} et inflige {degats} dégâts.");
        cible.PointsDeVie -= degats;
    }
}

class Program
{
    static void Main()
    {
        Personnage aragorn = new Personnage("Aragorn", "Guerrier", 100, 15, 10);
        Personnage legolas = new Personnage("Legolas", "Archer",    80, 18,  7);
```

```
        aragorn.Afficher();
        legolas.Afficher();

        Console.WriteLine("\n=== Combat ===");
        aragorn.Attaquer(legolas);
        legolas.Attaquer(aragorn);

        Console.WriteLine("\n=== Résultat ===");
        aragorn.Afficher();
        legolas.Afficher();
    }
}
```

Résultat :

```
═══ Début du combat ═══

[Guerrier] Aragorn – PV : 100 – ATQ : 15 – DEF : 10 – ✓ Vivant
[Archer] Legolas – PV : 80 – ATQ : 18 – DEF : 7 – ✓ Vivant
[Nain] Gimli – PV : 120 – ATQ : 12 – DEF : 14 – ✓ Vivant

═══ Tour 1 ═══
Aragorn attaque Legolas et inflige 8 dégâts.
Legolas attaque Gimli et inflige 4 dégâts.
Gimli attaque Aragorn et inflige 2 dégâts.

═══ Fin du tour ═══
[Guerrier] Aragorn – PV : 98 – ATQ : 15 – DEF : 10 – ✓ Vivant
[Archer] Legolas – PV : 72 – ATQ : 18 – DEF : 7 – ✓ Vivant
[Nain] Gimli – PV : 116 – ATQ : 12 – DEF : 14 – ✓ Vivant
```

7. Comparaison côte à côte

	PROCÉDURAL	ORIENTÉ OBJET
Données d'un personnage	Éparpillées dans 6 variables	Regroupées dans un objet
Logique d'attaque	Copiée à chaque attaque	Écrite une seule fois dans `Attaquer()`
Corriger le calcul de dégâts	Modifier 10 endroits	Modifier 1 seul endroit
Ajouter un 4e personnage	Créer 6 nouvelles variables	`new Personnage( ... )`
Passer un personnage à une fonction	6 paramètres	1 seul paramètre de type `Personnage`
Protéger les données	Impossible	`private set` empêche la modification directe



8. Ce que tu dois retenir de ce module

Les 3 idées clés :

1. **Une classe regroupe les données ET les comportements** qui vont naturellement ensemble.
2. **Un objet est une instance d'une classe** — chaque personnage est un objet indépendant.
3. **La logique métier est écrite une seule fois** dans la classe — pas copiée-collée partout.

Exercices

Ex. 1 — Lire et comprendre

Dans Rider, crée un nouveau projet **Console Application (.NET)** (`File > New Solution > Console Application`), colle le code de la section 6 et lance-le avec `Run` (▶) ou `Shift+F10` . Modifie les valeurs initiales des personnages et observe comment le résultat change. Ajoute un 4e personnage (`Gandalf` , `Mage` , 70, 25, 5) et fais-le participer au combat.

Ex. 2 — Étendre la classe

Ajoute une méthode `Soigner(int soin)` qui augmente les PV d'un personnage **sans dépasser une valeur maximale**. Définis cette valeur maximale comme attribut (initialisé à la construction à la même valeur que les PV de départ).

Ex. 3 — Comparer (discussion en classe)

Dans le programme procédural avec tableaux (section 3), que se passe-t-il si l'utilisateur entre 10 personnages dont les noms sont saisies au clavier ? Écris les 5 premières lignes de ce programme en procédural, puis en orienté objet. Laquelle préfères-tu ? Pourquoi ?

UAA 14 — Module 1 — 6e TTR Informatique

Chapitre 1 – Classes et objets

> *Tu sais déjà programmer. Tu sais déclarer des variables, écrire des fonctions, utiliser des listes. Maintenant, on change de perspective. En programmation orientée objet, on ne pense plus en termes d'instructions, mais en termes d'objets — des entités autonomes qui savent ce qu'elles sont et ce qu'elles peuvent faire. C# est un langage conçu pour ça, et ce premier chapitre pose la brique fondamentale : la classe.*

Objectifs du chapitre

À la fin de ce chapitre, tu seras capable de :

- Expliquer la différence entre une **classe** et un **objet**
- Déclarer une classe en C# avec des **attributs** et des **méthodes**
- **Instancier** un objet à partir d'une classe
- Appeler les méthodes d'un objet instancié
- Stocker plusieurs objets dans une **liste**

---

5 notions-clés

#	NOTION	EN UNE PHRASE
1	**Classe**	Le plan de construction — le *moule* qui décrit à quoi ressemble un objet
2	**Objet**	Une instance concrète créée à partir d'une classe
3	**Attribut**	Une donnée propre à chaque objet (ex : la couleur d'une voiture)
4	**Méthode**	Une action que l'objet sait effectuer (ex : démarrer)
5	**Instanciation**	L'acte de créer un objet à partir d'une classe avec le mot-clé `new`

Classe = le moule, objet = la pièce

Imagine une usine automobile. L'ingénieur dessine un **plan de fabrication** pour la Peugeot 208 : couleur, nombre de portes, motorisation... Ce plan, c'est la **classe**.

Quand l'usine fabrique une vraie voiture à partir de ce plan, elle crée un **objet** (une instance). On peut fabriquer des dizaines de voitures à partir du même plan : chacune est un objet différent, mais elles partagent la même structure.

> ***Classe*** *= le plan (existe une seule fois dans le code)*
> ***Objet*** *= la voiture réelle (on peut en créer autant qu'on veut)*

Déclarer une classe en C#

```
class Voiture
{
    // Attributs : les données de chaque voiture
    public string Marque;
    public string Couleur;
    public int NombreDePortes;

    // Méthode : une action que la voiture peut effectuer
    public void Demarrer()
    {
        Console.WriteLine($"{Marque} démarre. Vroum !");
    }

    public void AfficherInfos()
    {
        Console.WriteLine($"Voiture : {Marque} | Couleur : {Couleur} | Portes : {NombreDePo
    }
}
```

Ce qu'il faut retenir :

- Le mot-clé `class` déclare une nouvelle classe
- Les **attributs** (`Marque` , `Couleur` , `NombreDePortes`) stockent les données de l'objet
- Les **méthodes** (`Demarrer()` , `AfficherInfos()`) définissent ce que l'objet peut faire
- Le mot-clé `public` signifie que ces membres sont accessibles depuis l'extérieur de la classe (on reviendra sur ce point au chapitre 2)

Instancier un objet

Pour créer un objet à partir de la classe, on utilise le mot-clé `new` :

```
// Création (instanciation) d'un objet de type Voiture
Voiture maVoiture = new Voiture();

// On donne des valeurs à ses attributs
maVoiture.Marque = "Peugeot";
maVoiture.Couleur = "Rouge";
maVoiture.NombreDePortes = 5;

// On appelle ses méthodes
maVoiture.AfficherInfos();   // Voiture : Peugeot | Couleur : Rouge | Portes : 5
maVoiture.Demarrer();        // Peugeot démarre. Vroum !
```

▸ Le point `.` (opérateur d'accès)

Pour accéder aux attributs ou aux méthodes d'un objet, on utilise le point `.` :

```
nomObjet.Attribut
nomObjet.Methode()
```

Plusieurs objets, une seule classe

C'est là que la POO devient puissante : on peut créer **autant d'objets qu'on veut** à partir du même plan.

```
// Trois voitures instanciées à partir de la même classe
Voiture voiture1 = new Voiture();
voiture1.Marque = "Peugeot";
voiture1.Couleur = "Rouge";
voiture1.NombreDePortes = 5;

Voiture voiture2 = new Voiture();
voiture2.Marque = "Renault";
voiture2.Couleur = "Bleu";
voiture2.NombreDePortes = 3;

Voiture voiture3 = new Voiture();
voiture3.Marque = "BMW";
voiture3.Couleur = "Noir";
voiture3.NombreDePortes = 5;

// Chaque objet est indépendant
voiture1.AfficherInfos();
voiture2.AfficherInfos();
voiture3.AfficherInfos();
```

Chaque objet possède ses **propres valeurs** pour chaque attribut. Modifier `voiture1.Couleur` n'affecte pas `voiture2` ni `voiture3`.

Stocker des objets dans une liste

En pratique, on travaille souvent avec de nombreux objets. On les stocke dans une `List<T>` :

```
List<Voiture> parking = new List<Voiture>();

// On crée et on ajoute chaque voiture
Voiture v1 = new Voiture();
v1.Marque = "Peugeot";
v1.Couleur = "Rouge";
v1.NombreDePortes = 5;
parking.Add(v1);

Voiture v2 = new Voiture();
v2.Marque = "Renault";
v2.Couleur = "Bleu";
v2.NombreDePortes = 3;
parking.Add(v2);

// On parcourt la liste avec foreach
foreach (Voiture v in parking)
{
    v.AfficherInfos();
}
```

Exercice guidé — Le garage

Objectif

Créer une application console C# qui gère un garage de voitures.

Étape 1 — Crée la classe

Crée un nouveau projet console C# dans Rider. Ajoute un fichier `Voiture.cs` et déclare la classe `Voiture` avec les attributs et méthodes vus dans le cours.

Étape 2 — Instancie 3 voitures

Dans `Program.cs`, crée 3 voitures avec des caractéristiques différentes et affiche leurs informations.

▸ Étape 3 — Utilise une liste

Stocke tes 3 voitures dans une `List<Voiture>` et affiche-les toutes avec un `foreach`.

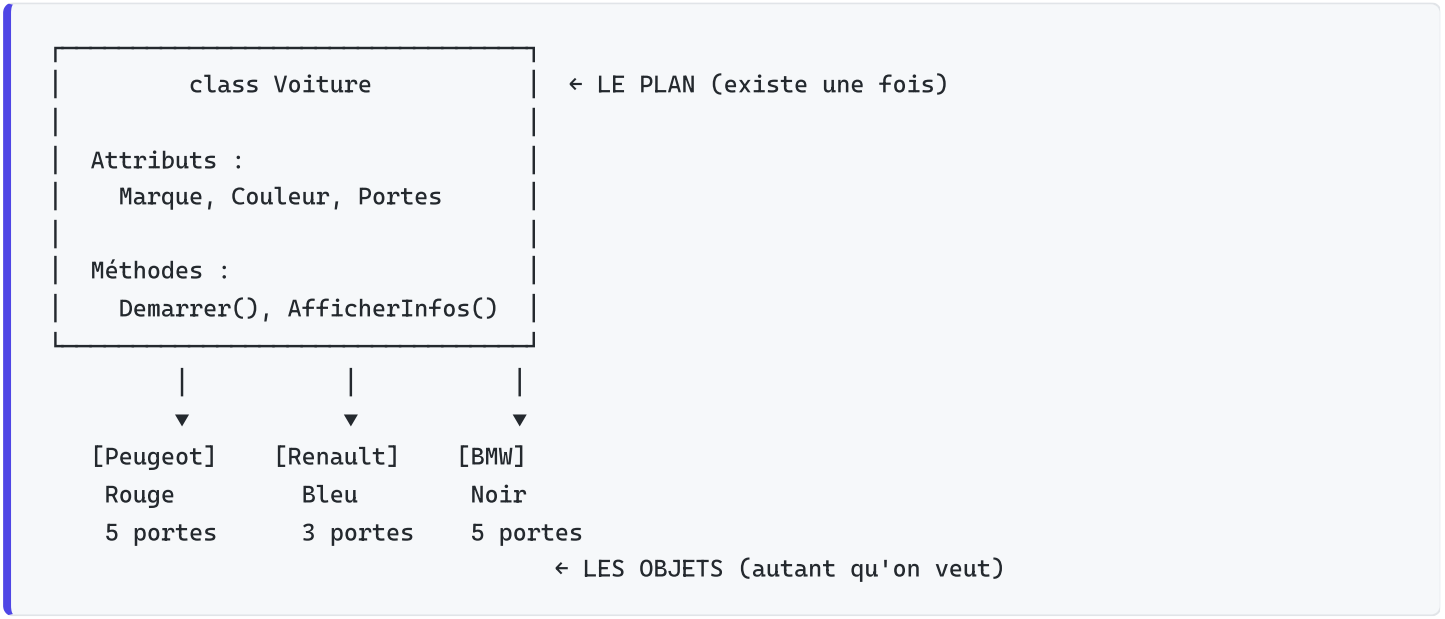
▸ Étape 4 — Ajoute une méthode

Ajoute une méthode `Klaxonner()` à la classe. Elle affiche : `{Marque} fait : Tuuut !`. Appelle-la sur chaque voiture de ta liste.

▸ Étape 5 — Défi

Ajoute un attribut `int Kilometrage` à ta classe. Crée une méthode `Rouler(int km)` qui **augmente** le kilométrage de la voiture et affiche un message.

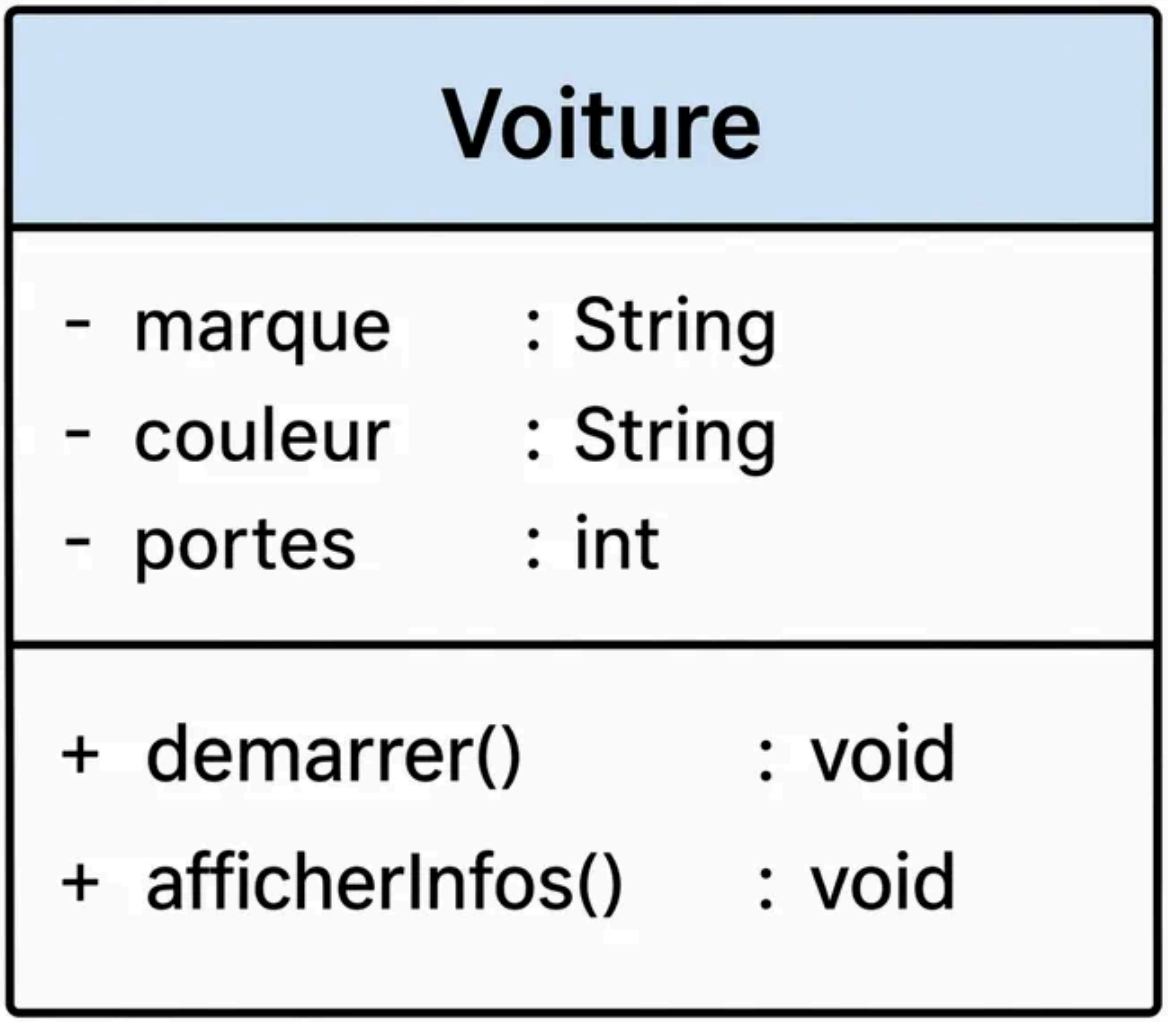
Schéma récapitulatif



Lire le schéma d'une classe

Quand on conçoit une application orientée objet, on dessine souvent les classes sous forme d'un schéma avant d'écrire le moindre code. C'est plus rapide qu'un programme, et ça permet de réfléchir à la structure sans se perdre dans la syntaxe.

Voici le schéma correspondant à notre classe `Voiture` :



Le schéma est découpé en **trois compartiments** :

En-tête — le nom de la classe, mis en évidence. C'est le nom qu'on utilisera dans le code avec le mot-clé `class`.

Attributs — chaque ligne indique un attribut avec son type.

Le signe `-` signifie que l'attribut est *privé* : on ne peut pas y accéder directement depuis l'extérieur de la classe. (On verra pourquoi c'est important au chapitre 2.)

Méthodes — chaque ligne indique une méthode avec son type de retour.

Le signe `+` signifie que la méthode est *publique* : n'importe qui peut l'appeler depuis l'extérieur.

> `-` ***= privé*** · `+` ***= public*** · `: type` ***= ce que ça contient ou retourne***

Ce schéma se lit donc comme ça :

Ce qu'on verra au chapitre suivant

Pour l'instant, nos attributs sont `public` : n'importe qui peut les modifier directement. C'est un problème — rien n'empêche d'écrire `voiture1.NombreDePortes = -12`. Au chapitre 2, on découvrira les mots-clés `public` et `private` : la première brique pour protéger les données d'un objet. Les chapitres 3 et 4 verront ensuite comment rouvrir un accès contrôlé via les *getters/setters* puis les *propriétés* C#.

Exercices – Classes et objets

> *Exercices de modélisation*

Consigne

Choisis **3 classes** dans la liste ci-dessous.
Pour chacune, tu dois :

1. Dessiner le **schéma** de la classe (nom, attributs avec types, méthodes avec type de retour) **SUR PAPIER**
2. **Coder la classe** en C# avec ses attributs et ses méthodes
3. Dans `Program.cs`, **instancier 2 objets** de cette classe, leur donner des valeurs et appeler chaque méthode

---

Les 10 classes disponibles

1. Livre
Une bibliothèque veut modéliser ses livres. Chaque livre possède un titre, un auteur et un nombre de pages. On doit pouvoir afficher sa fiche complète et simuler sa lecture (affiche un message indiquant qu'on est en train de lire le livre).

---

2. Compte bancaire
Une banque gère des comptes clients. Chaque compte a un titulaire, un numéro de compte et un solde. On doit pouvoir déposer de l'argent, retirer de l'argent (sans passer en négatif) et afficher le solde actuel.

---

3. Étudiant
Une école suit les résultats de ses élèves. Chaque étudiant a un nom, une classe et un total de points. On doit pouvoir ajouter des points à son total et afficher son bulletin (nom, classe, points).

---

4. Lampe connectée
Une application domotique contrôle des lampes. Chaque lampe a une couleur, une intensité (0 à 100) et un état (allumée ou éteinte). On doit pouvoir l'allumer, l'éteindre et changer sa couleur.

---

5. Produit
Une boutique gère son stock. Chaque produit a un nom, un prix et une quantité en stock. On doit pouvoir vendre une unité (diminue le stock), réapprovisionner (augmente le stock) et afficher la fiche du produit.

---

6. Animal de compagnie
Un refuge pour animaux tient un registre. Chaque animal a un nom, une espèce et un âge. On doit pouvoir lui faire faire du bruit (affiche un son selon l'espèce) et afficher ses informations.

---

7. Film
Un cinéma catalogue ses films. Chaque film a un titre, un réalisateur, une durée en minutes et un genre. On doit pouvoir afficher sa fiche et savoir s'il est long (plus de 120 minutes → retourne `true` ou `false`).

---

8. Message
Une application de messagerie gère des messages. Chaque message a un expéditeur, un destinataire, un contenu et un état (lu ou non lu). On doit pouvoir envoyer le message (affiche un résumé) et le marquer comme lu.

---

9. Joueur
Un jeu vidéo suit la progression de ses joueurs. Chaque joueur a un pseudo, un niveau et des points d'expérience (XP). On doit pouvoir lui faire gagner de l'XP et passer au niveau suivant (augmente le niveau, remet l'XP à zéro), et afficher ses stats.

---

10. Capteur météo
Un réseau de capteurs surveille la météo. Chaque capteur a une localisation, une température et un taux d'humidité. On doit pouvoir déclencher une mesure (génère des valeurs aléatoires) et afficher les données relevées.

▸ Critères d'évaluation

CRITÈRE	POINTS
Schéma correct (nom, attributs typés, méthodes)	3 × 2 pts
Classe C# avec attributs et méthodes	3 × 3 pts
Instanciation et appel des méthodes dans `Program.cs`	3 × 2 pts
Code compilable sans erreur	3 pts
Total	**24 pts**

Chapitre 2 – Attributs et visibilité

> *Au Chapitre 1, n'importe qui pouvait écrire `voiture.NombreDePortes = -12` sans que personne ne proteste. Le compilateur accepte, l'objet se retrouve dans un état impossible. C'est le signe que nos attributs ne sont pas protégés. Dans ce chapitre, on découvre les deux mots-clés qui posent les bases de la solution : `public` et `private`.*

La fin du Chapitre 1 a laissé un trou : `voiture.NombreDePortes = -12` est accepté sans broncher. Ce chapitre regarde ce trou de plus près, et pose la première pierre pour le boucher.

🎯 Objectifs

À la fin de ce chapitre, tu seras capable de :

1. Expliquer pourquoi des attributs accessibles à tous posent problème.
2. Utiliser les mots-clés `public` et `private` correctement.
3. Reconnaître l'erreur de compilation qui apparaît quand on tente d'accéder à un attribut privé depuis l'extérieur.
4. Énoncer la règle de l'encapsulation : *attributs privés, accès contrôlé*.

Partie 1 — Un objet sans défense

Reprends la classe `Voiture` du Chapitre 1 :

```
class Voiture
{
    public string Marque;
    public string Couleur;
    public int NombreDePortes;
}
```

Et son utilisation :

```
Voiture maVoiture = new Voiture();
maVoiture.Marque = "Peugeot";
maVoiture.NombreDePortes = 5;     // Cohérent.
maVoiture.NombreDePortes = -12;   // Absurde. Mais accepté.
maVoiture.NombreDePortes = 9999;  // N'importe quoi. Accepté aussi.
```

Le compilateur ne bronche pas. À l'exécution, rien n'explose. **L'objet est dans un état impossible, et personne ne le sait.**

Ce n'est pas spécifique à `Voiture`. Le même problème touche n'importe quelle classe avec des attributs `public` :

```
Personnage hero = new Personnage();
hero.Pv     = -9999;   // Personne ne proteste.
hero.Niveau = -42;     // Idem.
```

> 🧠 ***L'idée à saisir :*** *la responsabilité de protéger les données d'un objet doit être* ***à l'intérieur de cet objet****. C'est à l'objet de refuser ce qui n'a pas de sens — pas au code appelant de penser à vérifier à chaque ligne.*

Pour ça, il faut commencer par **fermer la porte de l'extérieur**. C'est le rôle de la visibilité.

Partie 2 — Les mots-clés `public` et `private`

En C#, chaque membre d'une classe (attribut, méthode, propriété…) a une **visibilité**. Elle décide d'où ce membre peut être lu ou modifié.

Deux mots-clés à connaître pour démarrer :

MOT-CLÉ	ACCESSIBLE DEPUIS	USAGE TYPIQUE
`public`	N'importe où dans le projet	Méthodes, propriétés (l'interface de l'objet)
`private`	Uniquement à l'intérieur de la classe	Attributs internes (les données privées)

> 💡 ***Si on ne met rien****, C# considère le membre comme* `private` *par défaut. On préfère quand même l'écrire explicitement — c'est plus lisible.*

▸ Rendons les attributs de `Voiture` privés

```
class Voiture
{
    private string marque;          // ← private
    private string couleur;         // ← private
    private int    nombreDePortes;  // ← private
}
```

> ℹ️ ***Convention de nommage*** *: les attributs privés s'écrivent traditionnellement en camelCase (première lettre minuscule) :* `nombreDePortes`*. Les propriétés et méthodes publiques s'écrivent en PascalCase (première lettre majuscule) :* `NombreDePortes`*,* `Demarrer()`*. On verra l'intérêt de cette distinction au chapitre suivant.*

Partie 3 — Conséquence : le code d'avant ne compile plus

Reprends maintenant ton `Program.cs` du Chapitre 1 :

```
Voiture maVoiture = new Voiture();
maVoiture.marque = "Peugeot";       // ❌ Erreur de compilation
maVoiture.couleur = "Rouge";        // ❌
maVoiture.nombreDePortes = 5;       // ❌
```

Dans Rider, tu vois apparaître une erreur en rouge sur chacune de ces lignes :

```
'Voiture.marque' is inaccessible due to its protection level
```

C'est exactement ce qu'on voulait. Le compilateur empêche maintenant n'importe qui de modifier les données de la voiture en passant par-dessus la classe.

Sauf que… on a un nouveau problème : **on ne peut plus rien faire avec notre voiture**. On a fermé la porte, mais on n'a pas encore la clé.

> ⚠️ ***Erreur classique*** *: passer en* `private` *sans prévoir comment l'extérieur va lire et écrire les données. La classe devient inutilisable. Si tu casses tout ton* `Program.cs` *en mettant* `private` *partout, c'est normal — la solution est dans les deux chapitres suivants.*

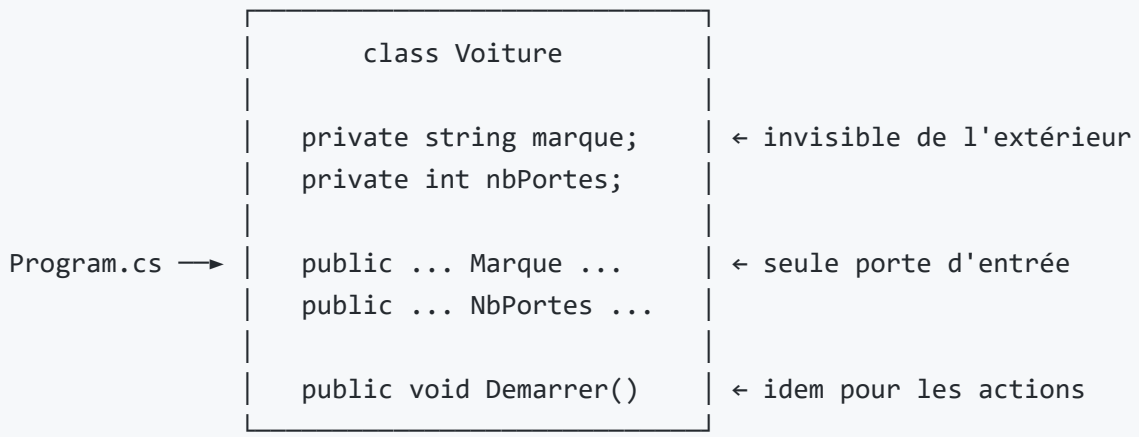
Partie 4 — La règle de l'encapsulation

L'encapsulation, c'est l'idée que **les données d'un objet sont cachées à l'intérieur de cet objet, et qu'on n'y accède que par des points d'entrée contrôlés**.

La règle de base, à retenir une fois pour toutes :

> 🧠 ***Les attributs sont*** `private` ***. Les points d'accès sont*** `public` ***.***

Schématiquement :



Tout ce qui dépasse à l'extérieur de la classe, c'est **l'interface publique** de l'objet. C'est elle qui décide ce qu'on peut faire avec lui, et dans quelles limites.

Les méthodes `Demarrer()`, `AfficherInfos()` du Chapitre 1 sont déjà `public` — c'est normal, elles sont faites pour être appelées depuis l'extérieur. Ce qui change ici, c'est que **les données** elles, deviennent `private`.

🧪 Exercices

Exercice 1 — Casser et observer

Reprends ton projet du Chapitre 1. Passe **tous les attributs de `Voiture` en `private`**, puis essaie de lancer le programme.

1. Note les erreurs de compilation affichées par Rider.
2. Compte combien de lignes de `Program.cs` sont concernées.
3. Conclus : pourquoi est-ce une **bonne nouvelle**, même si le programme ne tourne plus pour l'instant ?

Exercice 2 — Identifier les attributs à protéger

Pour chacune des classes ci-dessous, indique :

- Quels attributs devraient être `private` (presque toujours : tous).
- Pour chaque attribut, donne **un exemple de valeur absurde** qui devrait être refusée si quelqu'un essayait de l'affecter directement.

CLASSE	ATTRIBUTS
Personnage	`nom`, `pv`, `pvMax`, `niveau`
CompteBancaire	`titulaire`, `solde`
LampeConnectee	`couleur`, `intensite` (0 à 100), `allumee`
Produit	`nom`, `prix`, `quantiteEnStock`

▸ Exercice 3 — Lecture d'erreur

Tu écris ce code dans Rider :

```
class Compte
{
    private double solde;
}

// Dans Program.cs :
Compte c = new Compte();
c.solde = 1000;
Console.WriteLine(c.solde);
```

Rider affiche une erreur. **Sans la lire** :

1. Écris la phrase d'erreur que tu **devines** voir apparaître.
2. Sur quelles lignes apparaît-elle ?
3. Compile et compare avec ce que Rider montre réellement.

✍️ À retenir

- Des attributs `public` exposent un objet à des **affectations absurdes** que rien n'empêche.
- `public` = accessible partout · `private` = accessible uniquement dans la classe.
- La règle de l'encapsulation : **attributs** `private`, **interface** `public`.
- Mettre `private` casse temporairement le code appelant — c'est attendu. Il faut maintenant **ouvrir des points d'accès contrôlés** : c'est l'objet des chapitres 3 et 4.

Suite

On a fermé la porte. Maintenant, il faut une serrure et une clé : un moyen de **lire et d'écrire** ces attributs privés depuis l'extérieur, mais **en gardant le contrôle**. La solution la plus classique, qu'on rencontre dans tous les langages objet (Java, PHP, Python…), s'appelle les **getters** et les **setters**. C'est l'objet du chapitre suivant.

Chapitre 3 – Getters et setters

> *Au Chapitre 2 on a rendu les attributs privés — et cassé tout le code qui les utilisait. Il faut maintenant rouvrir des points d'accès, mais de manière contrôlée. La solution classique, qu'on retrouve dans tous les langages objet, consiste à fournir une méthode pour lire (le **getter**) et une méthode pour écrire (le **setter**).*

*Tu as vu au chapitre précédent que `private` ferme l'accès direct aux attributs. Cette barrière est utile — mais il faut quand même un moyen de lire et d'écrire ces données depuis l'extérieur. Les **getters** et **setters** sont la première réponse à ce besoin.*

🎯 Objectifs

À la fin de ce chapitre, tu seras capable de :

1. Écrire un **getter** : une méthode qui *retourne* la valeur d'un attribut privé.
2. Écrire un **setter** : une méthode qui *affecte* une nouvelle valeur à un attribut privé.
3. Ajouter de la **validation** dans un setter pour refuser les valeurs absurdes.
4. Reconnaître que cette solution est *verbeuse* — ce qui justifiera la syntaxe plus courte du chapitre suivant.

Partie 1 — La porte fermée du Chapitre 2

Reprenons l'exemple du `Personnage` :

```
class Personnage
{
    private string nom;
    private int    pv;
    private int    pvMax;
    private int    niveau;
}
```

Avec ces attributs `private`, le code suivant **ne compile plus** :

```
Personnage hero = new Personnage();
hero.nom = "Aragorn";   // ❌ inaccessible
hero.pv  = 100;         // ❌ inaccessible
```

Pour que `Program.cs` puisse encore travailler avec un personnage, il faut fournir des **méthodes publiques** qui font le pont entre l'extérieur et les attributs internes. C'est tout ce qu'est un getter ou un setter : une méthode `public` qui touche un attribut `private`.

Partie 2 — Le getter : lire une donnée

Un **getter** est une méthode qui **renvoie** la valeur d'un attribut. Par convention, son nom commence par `Get` suivi du nom de l'attribut.

```
class Personnage
{
    private int pv;

    public int GetPv()
    {
        return pv;
    }
}
```

Côté `Program.cs` :

```
Personnage hero = new Personnage();
Console.WriteLine(hero.GetPv());   // 0 (la valeur par défaut d'un int)
```

- `public` : la méthode est appelable depuis l'extérieur.
- `int` : c'est le type de ce que la méthode renvoie (un entier, comme l'attribut).
- `GetPv()` : pas de paramètre — on demande juste à lire.
- `return pv;` : on renvoie l'attribut privé.

> 💡 ***Pourquoi écrire une méthode juste pour renvoyer un attribut ?*** *Tant qu'elle ne fait que `return`, en effet elle n'apporte presque rien. Mais elle a un avantage caché : si un jour la lecture devient plus complexe (par exemple lire depuis un fichier, ou recalculer la valeur), on n'a **qu'un seul endroit** à modifier. Le code appelant continue d'appeler `hero.GetPv()` sans rien changer.*

Partie 3 — Le setter : écrire une donnée

Un **setter** est une méthode qui **affecte** une nouvelle valeur à un attribut. Son nom commence par `Set` suivi du nom de l'attribut. Il prend un paramètre — la nouvelle valeur — et ne renvoie rien (`void`).

```
public void SetPv(int valeur)
{
    pv = valeur;
}
```

Côté `Program.cs` :

```
hero.SetPv(100);
Console.WriteLine(hero.GetPv());   // 100
```

- `void` : la méthode ne renvoie rien.
- `int valeur` : le paramètre — la valeur qu'on essaie d'affecter.
- `pv = valeur;` : on assigne effectivement.

À ce stade, ce setter ne fait que recopier la valeur — il n'apporte pas grand-chose de plus qu'un attribut `public` aurait fait.

Sauf que. Maintenant qu'on contrôle l'écriture, on peut y ajouter ce qu'on veut.

Partie 4 — La vraie raison : la validation

Voici l'intérêt majeur du setter. Avant d'affecter la valeur, on peut la **vérifier** et la **corriger** si nécessaire.

```
public void SetPv(int valeur)
{
    if (valeur < 0)     valeur = 0;       // pas de PV négatifs
    if (valeur > pvMax) valeur = pvMax;   // pas plus que le max

    pv = valeur;
}
```

Maintenant, regarde ce qui se passe :

```
hero.SetPvMax(100);
hero.SetPv(120);     // demande 120, mais sera plafonné à 100
hero.SetPv(-9999);   // demande -9999, mais sera ramené à 0
Console.WriteLine(hero.GetPv());   // 0
```

Le code appelant peut bien essayer ce qu'il veut. L'objet refuse silencieusement, et reste dans un état cohérent. C'est précisément ce qu'on voulait depuis le Chapitre 2.

> 🧠 ***À retenir :*** *un setter sans validation, c'est un attribut public déguisé. Tout l'intérêt est dans le code qu'on peut placer **avant** l'affectation.*

▸ Quelques validations utiles

ATTRIBUT	VALIDATION TYPIQUE
`pv`	`valeur < 0` → 0 ; `valeur > pvMax` → pvMax
`niveau`	`valeur < 1` → 1
`nom`	`valeur == ""` ou `null` → "Inconnu"
`intensiteLampe`	`valeur < 0` → 0 ; `valeur > 100` → 100
`solde` (compte)	refuser un retrait qui passerait en négatif

Partie 5 — La classe `Personnage` au complet

Voici à quoi ressemble la classe avec tous ses getters et setters :

```
class Personnage
{
    // --- Attributs privés ---
    private string nom;
    private int    pv;
    private int    pvMax;
    private int    niveau;

    // --- Getters ---
    public string GetNom()    { return nom; }
    public int    GetPv()     { return pv; }
    public int    GetPvMax()  { return pvMax; }
    public int    GetNiveau() { return niveau; }

    // --- Setters avec validation ---
    public void SetNom(string valeur)
    {
        if (valeur == "" || valeur == null) valeur = "Inconnu";
        nom = valeur;
    }

    public void SetPv(int valeur)
    {
        if (valeur < 0)     valeur = 0;
        if (valeur > pvMax) valeur = pvMax;
        pv = valeur;
    }

    public void SetPvMax(int valeur)
    {
        if (valeur < 1) valeur = 1;
        pvMax = valeur;
    }

    public void SetNiveau(int valeur)
    {
        if (valeur < 1) valeur = 1;
        niveau = valeur;
    }
}
```

Et l'utilisation côté `Program.cs` :

```
Personnage hero = new Personnage();
hero.SetNom("Aragorn");
hero.SetPvMax(100);
hero.SetPv(100);
hero.SetNiveau(5);

Console.WriteLine($"{hero.GetNom()} - Niv.{hero.GetNiveau()} - {hero.GetPv()}/{hero.GetPvMa
// Aragorn - Niv.5 - 100/100 PV
```

Partie 6 — Le prix à payer : c'est verbeux

Regarde bien la classe ci-dessus. Pour **4 attributs**, on a écrit **8 méthodes**. Pour 6 attributs, ce serait 12 méthodes. Et chaque utilisation devient `hero.GetX()` au lieu de `hero.X` — plus long à lire, plus long à taper.

C'est la grande limite de cette approche :

> ⚠️ ***Les getters/setters explicites alourdissent énormément le code.*** *Une classe qui devrait tenir en 15 lignes en fait 60.*

Tous les langages objet ont ce problème. Java, PHP, Python… acceptent cette verbosité comme un mal nécessaire. C# a fait un choix différent : **proposer une syntaxe plus courte qui fait la même chose**.

C'est l'objet du chapitre suivant : les **propriétés**.

🧪 Exercices

Exercice 1 — Reprendre Voiture

Reprends ta classe `Voiture` du Chapitre 1. Passe tous les attributs en `private` et fournis pour chacun un getter et un setter.

Ajoute **au moins une validation** :

- `NombreDePortes` : entre 2 et 5 (sinon ramener à 5 par défaut).
- `Couleur` : refuser une chaîne vide (par défaut "Inconnue").

Teste depuis `Program.cs` :

```
maVoiture.SetNombreDePortes(-12);
Console.WriteLine(maVoiture.GetNombreDePortes());   // doit afficher 5
```

Exercice 2 — Le compte bancaire

Crée une classe `CompteBancaire` avec :

- `private string titulaire`
- `private double solde`

Et les méthodes suivantes :

MÉTHODE	COMPORTEMENT
`GetTitulaire()`	retourne le nom du titulaire
`GetSolde()`	retourne le solde
`SetTitulaire(string)`	refuse une chaîne vide
`Deposer(double montant)`	ajoute au solde **si le montant est positif**
`Retirer(double montant)`	retire du solde **si le solde reste ≥ 0** après l'opération

Note : `Deposer` et `Retirer` ne sont **pas** des setters classiques — ce sont des méthodes métier. Mais elles touchent l'attribut privé `solde` exactement de la même façon : avec validation. C'est une des forces de l'encapsulation — tout passage par la classe est l'occasion de contrôler.

▸ Exercice 3 — Compter les lignes

Pour la classe `Personnage` complète de la Partie 5 :

1. Compte le nombre de lignes utiles (sans commentaires, sans lignes vides).
2. Si tu ajoutes 2 attributs supplémentaires (`force` , `agilite`), combien de lignes en plus ?
3. Que penses-tu de ce ratio code-utile / code-administratif ?

✍️ À retenir

- Un **getter** est une méthode qui renvoie un attribut privé : `public int GetX() { return x; }` .
- Un **setter** est une méthode qui affecte un attribut privé : `public void SetX(int v) { ... x = v; }` .
- Le **vrai intérêt** est la **validation** dans le setter — refuser ou corriger les valeurs absurdes.
- Cette approche fonctionne, mais elle est **verbeuse** : 2 méthodes par attribut, et le code appelant devient lourd.
- C# propose mieux : les propriétés (chapitre suivant).

Suite

On a maintenant des données privées et des accès contrôlés — mais le code paie ça en verbosité. Au chapitre suivant, on découvre **les propriétés C#** : une syntaxe spécifique au langage qui fait exactement ce que font les getters/setters, en deux ou trois fois moins de lignes, et avec une utilisation presque transparente côté `Program.cs` .

Chapitre 4 – Les propriétés C#

> *Les propriétés sont la version moderne et concise des getters/setters du Chapitre 3. Elles se comportent comme des attributs `public` pour le code appelant, mais elles cachent à l'intérieur tout le contrôle qu'on souhaite. C# en fait un usage si systématique qu'on ne voit presque jamais de `GetX()` / `SetX()` dans un vrai projet C#.*

*Tu as vu au Chapitre 3 que les getters/setters fonctionnent mais qu'ils sont lourds : 8 méthodes pour 4 attributs, et le code appelant écrit `hero.GetPv()` partout. C# propose une syntaxe dédiée — la **propriété** — qui fait exactement la même chose, en deux fois moins de lignes, avec un usage transparent.*

🎯 Objectifs

À la fin de ce chapitre, tu seras capable de :

1. Écrire une **propriété** avec ses blocs `get` et `set`.
2. Utiliser le mot-clé `value` dans un `set`.
3. Choisir entre la **forme complète** (avec validation) et la **forme raccourcie** (`{ get; set; }`).
4. Limiter l'écriture avec `private set` (lecture publique, écriture interne).
5. Écrire une **propriété calculée** (un `get` sans `set`).
6. Encapsuler une liste interne dans une classe.

Partie 1 — Du getter/setter à la propriété

Reprends le getter/setter de `Pv` du chapitre précédent :

```
class Personnage
{
    private int pv;

    public int GetPv()
    {
        return pv;
    }

    public void SetPv(int valeur)
    {
        if (valeur < 0)     valeur = 0;
        if (valeur > pvMax) valeur = pvMax;
        pv = valeur;
    }
}
```

Voici la **même logique**, en C#, sous forme de propriété :

```
class Personnage
{
    private int pv;

    public int Pv
    {
        get
        {
            return pv;
        }
        set
        {
            if (value < 0)     value = 0;
            if (value > pvMax) value = pvMax;
            pv = value;
        }
    }
}
```

Trois choses à remarquer :

- La propriété s'écrit comme un attribut public (`public int Pv`) **suivi d'accolades** qui contiennent deux blocs : `get` et `set` .
- À l'intérieur du `set` , le mot-clé `value` désigne la valeur qu'on essaie d'assigner. C'est l'équivalent du paramètre du setter classique.
- L'attribut privé `pv` (minuscule) **est conservé** : c'est lui qui stocke réellement la donnée. La propriété `Pv` (majuscule) n'est qu'une *interface* qui l'expose proprement.

> 🧠 ***Convention :*** *attribut privé en camelCase (`pv`), propriété publique en PascalCase (`Pv`). C'est la convention C# universelle, et c'est précisément pour distinguer ces deux compagnons qu'elle existe.*

Partie 2 — L'usage côté `Program.cs`

C'est là que la propriété fait toute la différence. Du point de vue de celui qui utilise la classe, **une propriété s'utilise comme un attribut** :

```
Personnage hero = new Personnage();

hero.Pv = 100;       // appelle implicitement le set → stocke 100
hero.Pv = -9999;     // appelle implicitement le set → stocke 0 (validation)
hero.Pv = 50000;     // appelle implicitement le set → stocke pvMax (validation)

int actuels = hero.Pv;   // appelle implicitement le get → renvoie la valeur
Console.WriteLine(hero.Pv);
```

Pas de `Get`, pas de `Set`, pas de parenthèses. La protection est invisible de l'extérieur. **L'écriture est aussi simple qu'avec un attribut public — la sécurité est aussi solide qu'avec un setter classique.**

Partie 3 — La forme raccourcie

Très souvent, on n'a aucune validation à faire : on veut juste un attribut accessible en lecture et en écriture. Pour ce cas, C# propose une **forme raccourcie** : la propriété auto-implémentée.

```
public string Nom { get; set; }
```

Une seule ligne. Plus d'attribut privé à déclarer — C# en crée un automatiquement, en coulisses. Cette ligne est exactement équivalente à :

```
private string _nom;
public string Nom
{
    get { return _nom; }
    set { _nom = value; }
}
```

> ⚠️ ***Mais attention*** *: la forme raccourcie* ***ne permet aucune validation****. Dès que tu as besoin de refuser ou corriger une valeur, passe à la forme complète.*

▸ Variantes utiles

```
public string Classe { get; set; }              // lecture + écriture publiques
public int    Identifiant { get; private set; } // lecture publique, écriture interne uniqu
public bool   EstActif { get; }                 // lecture seule (rare, voir propriétés cal
```

La version avec `private set` est très utile : depuis l'extérieur, on ne peut que *lire* (par exemple `hero.Identifiant`), mais à l'intérieur de la classe, on peut écrire (par exemple à la création de l'objet).

Partie 4 — Les propriétés calculées

Une **propriété calculée** est une propriété qui **n'a pas d'attribut privé** dédié. Sa valeur est calculée à la volée à partir d'autres attributs. Elle n'a typiquement **pas de** `set` — on ne peut pas écrire dedans, seulement lire.

```
public bool EstVivant
{
    get { return pv > 0; }
}
```

Aucun attribut privé `estVivant` ; à chaque appel, on évalue l'expression `pv > 0`.

```
hero.Pv = 0;
Console.WriteLine(hero.EstVivant);   // False

hero.Pv = 50;
Console.WriteLine(hero.EstVivant);   // True
```

> 🧠 ***Personne ne peut écrire*** `hero.EstVivant = true` *: sans* `set`*, la propriété est en lecture seule. La seule manière de changer* `EstVivant`*, c'est de modifier* `Pv` *— ce qui est exactement ce qu'on veut.* ***La cohérence est garantie par construction.***

▸ D'autres exemples utiles

```
// Pourcentage de PV restants
public int PourcentagePv
{
    get { return (int)((double)pv / pvMax * 100); }
}

// Barre de vie textuelle : "[████████░░]"
public string BarreDeVie
{
    get
    {
        int nb = PourcentagePv / 10;
        return "[" + new string('█', nb) + new string('░', 10 - nb) + "]";
    }
}
```

▸ Syntaxe encore plus courte : `=>`

Pour les propriétés calculées d'une seule ligne, C# accepte une forme avec flèche :

```
public bool EstVivant => pv > 0;
public int  PourcentagePv => (int)((double)pv / pvMax * 100);
```

C'est strictement équivalent au bloc `get { return ...; }`, en beaucoup plus court.

Partie 5 — La classe `Personnage` au complet

Voici à quoi ressemble notre classe `Personnage`, encapsulée avec toutes les techniques vues ci-dessus :

```
class Personnage
{
    // --- Attributs privés ---
    private string nom;
    private int    pv;
    private int    pvMax;
    private int    niveau;

    // --- Propriété avec validation : Nom ---
    public string Nom
    {
        get { return nom; }
        set
        {
            if (value == "" || value == null) value = "Inconnu";
            nom = value;
        }
    }

    // --- Propriété avec validation : Pv (bornée entre 0 et PvMax) ---
    public int Pv
    {
        get { return pv; }
        set
        {
            if (value < 0)     value = 0;
            if (value > pvMax) value = pvMax;
            pv = value;
        }
    }

    // --- Propriété avec validation : PvMax (minimum 1) ---
    public int PvMax
    {
        get { return pvMax; }
        set
        {
            if (value < 1) value = 1;
            pvMax = value;
        }
    }

    // --- Propriété auto-implémentée : Classe ---
    public string Classe { get; set; }

    // --- Propriété avec validation : Niveau (minimum 1) ---
```

```
    public int Niveau
    {
        get { return niveau; }
        set
        {
            if (value < 1) value = 1;
            niveau = value;
        }
    }

    // --- Propriété calculée : EstVivant ---
    public bool EstVivant => pv > 0;

    // --- Propriété calculée : barre de vie textuelle ---
    public string BarreDeVie
    {
        get
        {
            int nb = pvMax == 0 ? 0 : pv * 10 / pvMax;
            return "[" + new string('', nb) + new string('', 10 - nb) + "]";
        }
    }
}
```

▸ Test dans `Program.cs`

```
Personnage hero = new Personnage();
hero.Nom    = "Aragorn";
hero.Classe = "Guerrier";
hero.PvMax  = 100;
hero.Pv     = 100;
hero.Niveau = 5;

Console.WriteLine($"{hero.Nom} ({hero.Classe}) — Niv.{hero.Niveau}");
Console.WriteLine($"PV : {hero.Pv}/{hero.PvMax} {hero.BarreDeVie}");
// Aragorn (Guerrier) — Niv.5
// PV : 100/100 [██████████]

hero.Pv = -9999;          // ramené à 0
Console.WriteLine(hero.BarreDeVie);
// [          ]
Console.WriteLine(hero.EstVivant);   // False
```

Compare le nombre de lignes avec le getter/setter du Chapitre 3 — et surtout compare la **lisibilité du code appelant**.

Partie 6 — Livrable : la classe `Inventaire`

Voici un exemple complet qui combine propriétés calculées et liste interne protégée.

La classe `Inventaire` contient une liste d'objets appartenant à un personnage. **La liste est `private`** : aucun code à l'extérieur ne peut la modifier directement. On passe obligatoirement par les méthodes prévues.

```
class Inventaire
{
    // La liste est privée et initialisée à la déclaration
    // (pas besoin de constructeur — on verra cette syntaxe au Chap. 6)
    private List<string> objets = new List<string>();

    // Capacité maximale (auto-implémentée, modifiable de l'extérieur)
    public int CapaciteMax { get; set; } = 10;

    // Propriété calculée : nombre d'objets dans l'inventaire
    public int Taille => objets.Count;

    // Propriété calculée : est-ce que l'inventaire est plein ?
    public bool EstPlein => objets.Count >= CapaciteMax;

    // --- Méthodes métier ---

    public void Ajouter(string objet)
    {
        if (EstPlein)
        {
            Console.WriteLine($"Inventaire plein ! Impossible d'ajouter {objet}");
            return;
        }
        objets.Add(objet);
        Console.WriteLine($"{objet} ajouté à l'inventaire.");
    }

    public void Retirer(string objet)
    {
        if (!objets.Contains(objet))
        {
            Console.WriteLine($"{objet} n'est pas dans l'inventaire.");
            return;
        }
        objets.Remove(objet);
        Console.WriteLine($"{objet} retiré de l'inventaire.");
    }

    public void Afficher()
    {
        if (Taille == 0)
        {
            Console.WriteLine("L'inventaire est vide.");
            return;
        }
```

```
        Console.WriteLine($"Inventaire ({Taille}/{CapaciteMax}) :");
        foreach (string o in objets)
        {
            Console.WriteLine($"  - {o}");
        }
    }
}
```

▸ **Utilisation**

```
Inventaire sac = new Inventaire();

sac.Ajouter("Épée longue");
sac.Ajouter("Potion de soin");
sac.Ajouter("Carte du donjon");

sac.Afficher();
// Inventaire (3/10) :
//   - Épée longue
//   - Potion de soin
//   - Carte du donjon

sac.Retirer("Potion de soin");
sac.Retirer("Arc elfique");   // n'existe pas → message d'erreur

Console.WriteLine($"Objets restants : {sac.Taille}");
// Objets restants : 2
```

💡 ***Observe que*** `Taille`, `EstPlein` *et* `CapaciteMax` *sont des propriétés, alors que* `Ajouter`, `Retirer` *et* `Afficher` *sont des méthodes. La règle pour choisir :* ***propriété pour exposer une donnée (lecture, écriture, calcul) ; méthode pour exécuter une action****. On formalisera tout ça au Chapitre 5.*

Partie 7 — Schéma récapitulatif

Voici comment les trois chapitres s'enchaînent :

```
CHAP. 1                CHAP. 2                CHAP. 3                CHAP. 4
_____________          _____________          _____________          _____________
public int Pv;     →   private int pv;    →   private int pv;    →   private int pv;

                                              public int GetPv()      public int Pv
                                              { return pv; }          {
                                                                          get { return pv
                                              public void SetPv(int v)    set
                                              {                           {
                                                  if (v < 0) v = 0;           if (value <
                                                  pv = v;                         value =
                                              }                               pv = value;
                                                                          }
                                                                      }

hero.Pv = -9999;       ❌ inaccessible        hero.SetPv(-9999);     hero.Pv = -9999;
(accepté, dégâts)      (compile error)       (stocke 0 silenc.)     (stocke 0 silenc.)
```

🧪 Exercices

Exercice 1 — Refondre `CompteBancaire`

Reprends ta classe `CompteBancaire` du Chapitre 3 (ex. 2). Convertis tes getters/setters en **propriétés**.

- `Titulaire` : propriété avec validation (refuser une chaîne vide).
- `Solde` : propriété en **lecture seule** depuis l'extérieur (`public double Solde { get; private set; }`). L'écriture passe uniquement par `Deposer` et `Retirer`.

Test : essaie `compte.Solde = 1_000_000;` depuis `Program.cs` — Rider doit refuser de compiler.

Exercice 2 — La lampe connectée

Crée une classe `LampeConnectee` avec :

- `Couleur` (string, propriété auto-implémentée).
- `Intensite` (int, propriété **avec validation** : bornée entre 0 et 100).
- `EstAllumee` (bool, propriété auto-implémentée).
- `Description` : propriété **calculée** qui renvoie une chaîne du genre : *"Lampe rouge — intensité 80% — allumée"* ou *"Lampe blanche — éteinte"* si la lampe n'est pas allumée.

Exercice 3 — Le produit en stock

Crée une classe `Produit` avec :

- `Nom` , `Prix` , `Quantite` (propriétés avec validations adaptées : prix ≥ 0, quantité ≥ 0).
- `ValeurStock` : propriété calculée = `Prix × Quantite` .

- `EnRupture` : propriété calculée = `true` si la quantité est à 0.

Bonus : ajoute une méthode `Vendre(int nombre)` qui diminue la quantité de `nombre` unités, en refusant si le stock est insuffisant.

✍️ À retenir

- Une **propriété** est la version C# du couple getter/setter, dans une syntaxe compacte.
- Dans un `set`, le mot-clé `value` représente la valeur que l'appelant cherche à affecter.
- **Forme complète** : pour valider, transformer, ou faire un calcul → blocs `get { ... }` / `set { ... }`.
- **Forme raccourcie** : `{ get; set; }` pour les propriétés sans validation — C# crée l'attribut privé tout seul.
- `private set` = lecture publique, écriture interne uniquement.
- Une **propriété calculée** n'a pas d'attribut privé : elle calcule sa valeur à chaque appel, et n'a pas de `set`.
- Une propriété **expose une donnée** ; une méthode **exécute une action**. Garde la distinction claire.

Suite

On a vu les **données** d'une classe en profondeur : attributs privés, propriétés, propriétés calculées. Il reste l'autre moitié — les **méthodes**. On les a utilisées sans jamais les formaliser : type de retour, paramètres, le mot-clé `this`, la méthode magique `ToString()`. C'est l'objet du chapitre suivant.

Chapitre 5 – Les méthodes

> *Depuis le Chapitre 1, on écrit des méthodes — `Demarrer()`, `AfficherInfos()`, `RecevoirDegats()`, `Ajouter()`… — sans jamais en avoir posé le vocabulaire. Ce chapitre formalise tout ça : anatomie d'une méthode, type de retour, paramètres, le mot-clé `this`, la méthode magique `ToString()`. À la fin, tu sais lire et écrire n'importe quelle méthode d'une classe C# avec assurance.*

*Une classe, c'est des **données** (attributs, propriétés) et des **actions** (méthodes). Les chapitres précédents ont creusé les données. On s'attaque maintenant à l'autre moitié.*

🎯 Objectifs

À la fin de ce chapitre, tu seras capable de :

1. Identifier les **5 parties** qui composent une méthode (visibilité, type de retour, nom, paramètres, corps).
2. Distinguer une méthode `void` d'une méthode qui **renvoie** une valeur.
3. Utiliser le mot-clé `this` pour désigner l'objet courant.
4. Redéfinir `ToString()` pour qu'un objet sache s'afficher tout seul.
5. Comprendre la différence entre une méthode **d'instance** et une méthode **statique** (`static`).

Partie 1 — Anatomie d'une méthode

Voici une méthode typique :

```
public int Soigner(int montant)
{
    int avant = pv;
    Pv += montant;
    int reellement = Pv - avant;
    return reellement;
}
```

Elle se découpe en **cinq parties** :

```
public      int      Soigner       (int montant)      { ... }
 │          │          │               │                 │
 │          │          │               │                 └─ Corps
 │          │          │               └─ Paramètres (entrées)
 │          │          └─ Nom (PascalCase, action)
 │          └─ Type de retour
 └─ Visibilité (public / private)
```

ÉLÉMENT	RÔLE
Visibilité	Qui peut l'appeler ? `public` depuis n'importe où, `private` depuis la classe
Type de retour	Ce que la méthode **renvoie** (`int` , `string` , `bool` , `void` si rien…)
Nom	Verbe à l'impératif, en PascalCase : `Demarrer` , `Soigner` , `Afficher`
Paramètres	Les valeurs d'entrée. Zéro, un, ou plusieurs. Chacun avec son type
Corps	Le bloc `{ ... }` qui exécute le travail

Partie 2 — Méthode `void` vs méthode qui renvoie

Deux grandes familles de méthodes :

▸ Les méthodes `void` — *je fais quelque chose*

Une méthode `void` exécute une action et **ne renvoie rien**. On l'utilise pour ses effets : afficher, modifier l'état de l'objet, écrire dans un fichier…

```
public void Afficher()
{
    Console.WriteLine($"{Nom} - {Pv}/{PvMax} PV");
}
```

Appel :

```
hero.Afficher();        // OK
int x = hero.Afficher();   // ❌ erreur : Afficher ne renvoie rien
```

▸ Les méthodes typées — *je calcule et je renvoie*

Une méthode avec un **type de retour** doit obligatoirement contenir une instruction `return` qui renvoie une valeur de ce type.

```
public int Soigner(int montant)
{
    int avant = pv;
    Pv += montant;
    return Pv - avant;   // nombre réel de PV récupérés
}
```

Appel :

```
int soin = hero.Soigner(30);
Console.WriteLine($"Tu as récupéré {soin} PV.");
```

> 💡 ***Comment choisir ?*** *Si ta méthode produit une **information** que l'appelant va utiliser ensuite, elle doit la **renvoyer**. Si elle ne fait qu'agir, elle est `void`. Quand tu hésites, demande-toi : est-ce que celui qui appelle cette méthode a besoin d'une réponse ?*

Partie 3 — Paramètres : valeurs et objets

Une méthode peut prendre **zéro, un ou plusieurs paramètres**, chacun avec son type. Ils s'écrivent en *camelCase*.

```
public void Deplacer(int x, int y)
{
    positionX = x;
    positionY = y;
}
```

▸ Passer un objet en paramètre

Le type d'un paramètre peut être une **classe** :

```
public void Attaquer(Personnage cible)
{
    int degats = Attaque - cible.Defense;
    if (degats < 0) degats = 0;

    cible.Pv -= degats;
    Console.WriteLine($"{Nom} attaque {cible.Nom} et inflige {degats} dégâts.");
}
```

Appel :

```
aragorn.Attaquer(legolas);
```

Du point de vue d'`aragorn`, `legolas` est passé sous le nom `cible`. Modifier `cible.Pv` modifie bien les PV de `legolas` — les objets sont passés **par référence** : on travaille sur l'original, pas sur une copie.

> ⚠️ ***Conséquence importante** : les méthodes qui prennent un objet en paramètre peuvent **modifier cet objet**. Sois conscient de ce que tu fais. Si tu ne veux pas qu'une méthode modifie son paramètre, ne le modifie pas dans le corps.*

Partie 4 — Le mot-clé `this`

`this` désigne **l'objet courant** — l'instance sur laquelle la méthode est appelée.

Quand tu écris `aragorn.Attaquer(legolas)`, à l'intérieur de la méthode `Attaquer` :

- `this` désigne `aragorn` (l'objet sur lequel on appelle la méthode).
- `cible` désigne `legolas` (le paramètre).

```
public void Attaquer(Personnage cible)
{
    Console.WriteLine($"{this.Nom} attaque {cible.Nom}.");
    // Strictement équivalent à : $"{Nom} attaque {cible.Nom}."
}
```

`this.Nom` et `Nom` font la même chose. Dans la plupart des cas, **on omet** `this` — il est implicite.

▸ Deux cas où `this` devient indispensable

1. LEVER UNE AMBIGUÏTÉ DE NOM

Si un **paramètre** porte le même nom qu'un **attribut**, le compilateur prend le paramètre par défaut. `this` permet de désigner explicitement l'attribut :

```
public void Renommer(string nom)
{
    this.nom = nom;
    //  ^----   ^----
    //   attribut  paramètre
}
```

Sans `this.`, la ligne `nom = nom;` ne ferait que se réaffecter le paramètre à lui-même — l'attribut ne changerait pas.

2. SE PASSER SOI-MÊME À UN AUTRE OBJET

Parfois, un objet doit se **passer en argument** à une méthode d'un autre objet :

```
public void RejoindreEquipe(Equipe equipe)
{
    equipe.Ajouter(this);   // « ajoute MOI à l'équipe »
}
```

Ici `this` représente le personnage courant. Sans `this`, on ne saurait pas comment se référer à lui-même.

Partie 5 — `ToString()` : la méthode universelle

Toutes les classes C# possèdent une méthode `ToString()` cachée qui renvoie une représentation textuelle de l'objet. Par défaut, elle renvoie le **nom complet de la classe** — pas très utile :

```
Personnage hero = new Personnage();
hero.Nom = "Aragorn";

Console.WriteLine(hero);   // → "Personnage" (par défaut, pas génial)
```

Mais on peut la **redéfinir** dans notre classe pour fournir un affichage personnalisé :

```
public override string ToString()
{
    return $"{Nom} ({Classe}) - Niv.{Niveau} - {Pv}/{PvMax} PV";
}
```

Maintenant :

```
Console.WriteLine(hero);
// Aragorn (Guerrier) - Niv.5 - 100/100 PV
```

`Console.WriteLine` appelle automatiquement `ToString()` sur l'objet — donc plus besoin de méthode `Afficher()` à part. L'interpolation `$"{hero}"` le fait aussi.

> 🧠 ***Le mot-clé*** `override` *signifie « je redéfinis une méthode qui existait déjà ». On verra exactement de qui elle vient quand on étudiera l'héritage. Pour l'instant, retiens juste qu'il est* ***obligatoire*** *pour redéfinir* `ToString()`.

▸ Bonnes pratiques pour `ToString()`

- Renvoyer une **chaîne courte et lisible**, sur une seule ligne.
- Inclure les attributs identifiants (nom, id…).
- Pas de saut de ligne (`\n`), pas d'effet de bord (pas d'écriture console à l'intérieur de `ToString` !).

Partie 6 — Méthodes d'instance vs méthodes statiques

Toutes les méthodes vues jusqu'ici sont des **méthodes d'instance** : on a besoin d'un objet pour les appeler.

```
Personnage hero = new Personnage();
hero.Afficher();   // on appelle Afficher SUR un objet
```

Il existe une autre catégorie : les **méthodes statiques**, déclarées avec le mot-clé `static`. Elles **n'appartiennent pas à un objet** — elles appartiennent à la classe elle-même.

```
class Outils
{
    public static int Maximum(int a, int b)
    {
        return a > b ? a : b;
    }
}

// Appel : pas de "new", on appelle DIRECTEMENT sur la classe
int gros = Outils.Maximum(5, 12);
```

Tu en as déjà rencontré sans le savoir :

```
Console.WriteLine("...");   // WriteLine est static
Math.Sqrt(16);              // Sqrt est static
int.Parse("42");            // Parse est static
```

C'est pour ça qu'on n'a jamais écrit `Console maConsole = new Console();`.

> 💡 ***Quand utiliser*** `static` ***?*** *Quand la méthode n'a pas besoin de l'état d'un objet particulier — typiquement des fonctions utilitaires (calculs, conversions, validation…). Si la méthode doit accéder à un attribut, elle ne peut* ***pas*** *être statique : il faut un objet.*

Partie 7 — La classe `Personnage` enrichie

Voici la classe avec son lot complet de méthodes :

```
class Personnage
{
    // --- Attributs et propriétés (rappel Chap. 4) ---
    private int pv;
    public string Nom    { get; set; }
    public string Classe { get; set; }
    public int    PvMax  { get; set; }
    public int    Niveau { get; set; }
    public int    Attaque { get; set; }
    public int    Defense { get; set; }

    public int Pv
    {
        get { return pv; }
        set
        {
            if (value < 0)     value = 0;
            if (value > PvMax) value = PvMax;
            pv = value;
        }
    }

    public bool EstVivant => pv > 0;

    // --- Méthode void : action sans retour ---
    public void Attaquer(Personnage cible)
    {
        int degats = Attaque - cible.Defense;
        if (degats < 0) degats = 0;

        cible.Pv -= degats;
        Console.WriteLine($"{Nom} attaque {cible.Nom} et inflige {degats} dégâts.");
    }

    // --- Méthode typée : renvoie le soin réel ---
    public int Soigner(int montant)
    {
        int avant = pv;
        Pv += montant;
        return pv - avant;
    }

    // --- Redéfinition de ToString ---
    public override string ToString()
    {
        string etat = EstVivant ? "vivant" : "mort";
```

```
        return $"{Nom} ({Classe}) - Niv.{Niveau} - {Pv}/{PvMax} PV - {etat}";
    }
}
```

▸ Test dans `Program.cs`

```
Personnage aragorn = new Personnage();
aragorn.Nom = "Aragorn";  aragorn.Classe = "Guerrier";
aragorn.PvMax = 100;  aragorn.Pv = 100;
aragorn.Attaque = 15; aragorn.Defense = 10; aragorn.Niveau = 5;

Personnage legolas = new Personnage();
legolas.Nom = "Legolas";  legolas.Classe = "Archer";
legolas.PvMax = 80;   legolas.Pv = 80;
legolas.Attaque = 18; legolas.Defense = 7; legolas.Niveau = 5;

Console.WriteLine(aragorn);
Console.WriteLine(legolas);

aragorn.Attaquer(legolas);
legolas.Attaquer(aragorn);

int soin = aragorn.Soigner(15);
Console.WriteLine($"{aragorn.Nom} récupère {soin} PV.");

Console.WriteLine(aragorn);
Console.WriteLine(legolas);
```

⚠️ ***Tu remarques*** *combien de lignes il faut pour initialiser* `aragorn` *et* `legolas` *? Sept lignes par personnage, et on* ***peut oublier d'en mettre une****, ce qui crée un objet à moitié initialisé. C'est le problème que va résoudre le constructeur, au chapitre suivant.*

🧪 Exercices

▸ Exercice 1 — Lire des méthodes

Pour chacune des signatures suivantes, dis **ce qu'elle renvoie** et **ce qu'elle prend en entrée** :

SIGNATURE	RENVOIE	PREND
`public void Allumer()`	…	…
`public bool EstAllume()`	…	…
`public int Compter(string mot, char lettre)`	…	…
`public string Resumer()`	…	…
`public void Combiner(Inventaire autre)`	…	…

▸ Exercice 2 — Ajouter des méthodes au Compte bancaire

Reprends ta classe `CompteBancaire` du Chapitre 4 et ajoute :

- `Transferer(double montant, CompteBancaire destinataire)` : retire du compte courant et dépose sur le destinataire — si le solde est suffisant.
- `EstRiche()` : méthode qui **renvoie** `true` si le solde dépasse 10 000 €.
- `ToString()` : retourne une chaîne du type *"Compte de Marie Dupont — solde : 1 245,00 €"*.

▸ Exercice 3 — Statique ou pas statique ?

Pour chacune des méthodes ci-dessous, indique si elle devrait être `static` ou **méthode d'instance**, et **pourquoi** :

MÉTHODE POSSIBLE	STATIC / INSTANCE ?
`Personnage.GenererPnj()` qui crée un personnage aléatoire	…
`aragorn.Soigner(20)`	…
`Math.Aleatoire(1, 100)`	…
`inventaire.Taille`	…
`Console.WriteLine( … )`	…

▸ Exercice 4 — Combat entre 3 personnages

Crée trois personnages (Aragorn, Legolas, Gimli) et fais-les se battre **chacun leur tour** :

- Tour 1 : Aragorn attaque Legolas, Legolas attaque Gimli, Gimli attaque Aragorn.
- Tour 2 : pareil.
- Affiche l'état de chacun après chaque tour avec `Console.WriteLine(personnage)`.
- Arrête le combat dès qu'un personnage tombe à 0 PV.

✍️ À retenir

- Une méthode = **visibilité + type de retour + nom + paramètres + corps**.
- `void` = la méthode ne renvoie rien. Sinon, le corps doit contenir un `return` du bon type.

- Les paramètres peuvent être des **valeurs** (`int` , `string` …) ou des **objets** (passés par référence).
- `this` désigne l'objet courant. Utile pour lever une ambiguïté ou se passer soi-même à une méthode d'un autre objet.
- `ToString()` se redéfinit avec `override` ; `Console.WriteLine(obj)` l'appelle automatiquement.
- **Méthode d'instance** : besoin d'un objet (`aragorn.Attaquer( ... )`). **Méthode** `static` : appartient à la classe, pas d'objet nécessaire (`Math.Sqrt( ... )`).

Suite

On sait maintenant écrire et lire à peu près n'importe quelle méthode. Reste un problème qu'on a pointé du doigt plusieurs fois : **créer un personnage prend 7 lignes**, et on peut en oublier une. Au chapitre suivant, le **constructeur** va régler ça en garantissant qu'un objet **naît directement dans un état valide**, en une seule ligne d'instanciation.

Chapitre 6 – Le constructeur

> *Jusqu'ici, créer un personnage prend sept lignes : `new Personnage()` puis six affectations. Et rien ne garantit qu'on n'en oublie pas une. Le **constructeur** est une méthode spéciale qui s'exécute au moment du `new` et qui prend en charge ces initialisations. Résultat : un objet naît toujours dans un état valide, en **une seule ligne**.*

Le `new` d'une classe est plus puissant que ce qu'on en a fait jusqu'à présent. Avec un constructeur, on peut donner à C# une recette précise pour fabriquer un objet — et garantir qu'aucun objet ne traîne avec des données manquantes ou absurdes.

🎯 Objectifs

À la fin de ce chapitre, tu seras capable de :

1. Reconnaître la **signature** d'un constructeur (nom = nom de la classe, pas de type de retour).
2. Écrire un constructeur qui initialise les attributs d'un objet.
3. Définir **plusieurs constructeurs** dans une même classe (surcharge).
4. Utiliser **les valeurs par défaut** pour rendre des paramètres facultatifs.
5. Chaîner deux constructeurs avec `this( ... )` pour éviter la duplication.

Partie 1 — Le problème : sept lignes pour un personnage

Reprends ton code d'instanciation du Chapitre 5 :

```
Personnage aragorn = new Personnage();
aragorn.Nom = "Aragorn";
aragorn.Classe = "Guerrier";
aragorn.PvMax = 100;
aragorn.Pv = 100;
aragorn.Attaque = 15;
aragorn.Defense = 10;
aragorn.Niveau = 5;
```

Sept lignes. Et il y a deux problèmes :

1. **Si tu en oublies une**, l'objet est dans un état incomplet sans qu'aucune erreur ne te le signale. Par exemple, oublier `aragorn.PvMax = 100;` casse silencieusement la validation des PV (puisque `Pv` se borne par `PvMax` qui vaut 0).
2. **Ce code est répétitif** : si tu instancies dix personnages, tu écris soixante-dix lignes presque identiques.

L'idée du constructeur, c'est de **regrouper toute cette initialisation à un seul endroit** : la classe elle-même.

Partie 2 — Anatomie d'un constructeur

Un **constructeur** est une méthode spéciale qui :

- Porte **exactement le nom de la classe** (`Personnage` pour la classe `Personnage`).
- N'a **pas de type de retour** (même pas `void`).
- Est exécutée **automatiquement** à chaque appel de `new` .

Voici un constructeur basique pour `Personnage` :

```
class Personnage
{
    // ... attributs et propriétés ...

    public Personnage(string nom, string classe, int pvMax)
    {
        Nom    = nom;
        Classe = classe;
        PvMax  = pvMax;
        Pv     = pvMax;   // on commence avec tous les PV
        Niveau = 1;
    }
}
```

Et l'utilisation devient :

```
Personnage aragorn = new Personnage("Aragorn", "Guerrier", 100);
```

Une seule ligne. L'objet est immédiatement utilisable, et il est impossible d'oublier une initialisation : le compilateur **exige** les trois paramètres.

> 🧠 ***Comment lire*** `new Personnage(...)` ***:***
>
> 1. *C# alloue un nouvel objet* `Personnage` *en mémoire.*
> 2. *Tous les attributs prennent leur valeur par défaut (* `0` , `null` , `false` *…).*
> 3. *C# appelle le* ***constructeur*** *avec les arguments fournis.*
> 4. *L'objet est renvoyé à l'appelant, prêt à l'emploi.*

Partie 3 — Le constructeur par défaut

Si tu **n'écris aucun constructeur** dans ta classe, C# en fabrique un implicitement, sans paramètre et sans corps. C'est ce qui te permettait, jusqu'au Chapitre 5, d'écrire `new Personnage()` sans avoir rien défini.

```
// Classe sans constructeur écrit
class Voiture
{
    public string Marque { get; set; }
}

// On peut quand même appeler :
Voiture v = new Voiture();   // ← constructeur par défaut implicite
```

Mais attention : dès que tu écris *au moins un* constructeur, C# **n'ajoute plus le constructeur par défaut**. Si tu veux conserver la possibilité d'instancier sans arguments, tu dois l'écrire explicitement.

```
class Personnage
{
    public Personnage(string nom, string classe, int pvMax) { ... }
}

// Désormais :
Personnage hero = new Personnage();   // ❌ erreur de compilation
Personnage hero = new Personnage("Aragorn", "Guerrier", 100);   // ✅
```

Partie 4 — Lever l'ambiguïté avec `this`

Tu vois souvent les constructeurs écrits comme ceci, avec un paramètre qui porte le même nom que l'attribut privé :

```
class Voiture
{
    private string marque;

    public Voiture(string marque)
    {
        this.marque = marque;
        //  ^----        ^----
        //   attribut     paramètre
    }
}
```

Sans `this.`, la ligne `marque = marque;` ne ferait que se réaffecter le paramètre à lui-même. Avec `this.`, on précise qu'on veut écrire **dans l'attribut** de l'objet courant.

C'est exactement l'usage de `this` vu au Chapitre 5. Les constructeurs en sont le terrain de jeu le plus fréquent.

Partie 5 — Plusieurs constructeurs (surcharge)

Une classe peut avoir **plusieurs constructeurs**, à condition qu'ils diffèrent par leurs paramètres (nombre ou type). Cette technique s'appelle la **surcharge**.

```
class Personnage
{
    // Constructeur complet
    public Personnage(string nom, string classe, int pvMax, int attaque, int defense)
    {
        Nom = nom;
        Classe = classe;
        PvMax = pvMax;
        Pv = pvMax;
        Attaque = attaque;
        Defense = defense;
        Niveau = 1;
    }

    // Constructeur simplifié (valeurs « moyennes » par défaut)
    public Personnage(string nom, string classe)
    {
        Nom = nom;
        Classe = classe;
        PvMax = 50;
        Pv = 50;
        Attaque = 10;
        Defense = 10;
        Niveau = 1;
    }

    // Constructeur vide (PNJ générique)
    public Personnage()
    {
        Nom = "Inconnu";
        Classe = "Aventurier";
        PvMax = 30;
        Pv = 30;
        Attaque = 5;
        Defense = 5;
        Niveau = 1;
    }
}
```

C# choisit automatiquement le bon constructeur selon les arguments fournis :

```
Personnage aragorn = new Personnage("Aragorn", "Guerrier", 100, 15, 10);  // → constructeur
Personnage paysan  = new Personnage("Otto", "Paysan");                    // → constructeur
Personnage inconnu = new Personnage();                                    // → constructeur
```

> ⚠️ ***Le piège de la duplication*** *: regarde les trois constructeurs ci-dessus. Le travail d'affectation se répète. Si tu ajoutes un attribut, tu dois modifier les trois. La section suivante montre comment éviter ça.*

Partie 6 — Chaîner avec `: this(...)`

Pour éviter la duplication, un constructeur peut **appeler un autre constructeur** de la même classe avec la syntaxe `: this( ... )`.

```
class Personnage
{
    // Constructeur principal — le seul qui contient la vraie logique
    public Personnage(string nom, string classe, int pvMax, int attaque, int defense)
    {
        Nom = nom;
        Classe = classe;
        PvMax = pvMax;
        Pv = pvMax;
        Attaque = attaque;
        Defense = defense;
        Niveau = 1;
    }

    // Constructeur intermédiaire → délègue au principal
    public Personnage(string nom, string classe)
        : this(nom, classe, 50, 10, 10)
    { }

    // Constructeur vide → délègue à l'intermédiaire
    public Personnage()
        : this("Inconnu", "Aventurier")
    { }
}
```

La syntaxe `: this( ... )` se lit *« avant d'exécuter mon corps, appelle ce constructeur-là avec ces arguments »*. La logique d'affectation se trouve à **un seul endroit** (le constructeur principal). Les autres ne sont que des « raccourcis ».

Partie 7 — Valeurs par défaut des paramètres

C# autorise une autre approche, encore plus concise : donner des **valeurs par défaut** aux paramètres du constructeur.

```
public Personnage(string nom = "Inconnu",
                  string classe = "Aventurier",
                  int    pvMax = 50,
                  int    attaque = 10,
                  int    defense = 10)
{
    Nom = nom;
    Classe = classe;
    PvMax = pvMax;
    Pv = pvMax;
    Attaque = attaque;
    Defense = defense;
    Niveau = 1;
}
```

Avec **un seul constructeur**, on couvre tous les cas :

```
Personnage aragorn = new Personnage("Aragorn", "Guerrier", 100, 15, 10);
Personnage paysan  = new Personnage("Otto", "Paysan");
Personnage inconnu = new Personnage();
```

▸ Et les arguments nommés ?

Tu peux préciser **seulement les paramètres qui t'intéressent**, en les nommant :

```
Personnage tank = new Personnage(nom: "Boromir", pvMax: 150, defense: 20);
// classe, attaque restent à leurs valeurs par défaut
```

> 💡 ***Surcharge ou valeurs par défaut ?*** *Les deux approches sont valides. La* ***surcharge*** *(avec `: this(...)`) est plus explicite et plus utile quand les constructeurs ont des logiques très différentes. Les* ***valeurs par défaut*** *sont plus concises quand on a juste des paramètres facultatifs. Dans la pratique, les valeurs par défaut suffisent dans 90 % des cas.*

Partie 8 — Initialiser une liste interne

Tu te souviens de la classe `Inventaire` du Chapitre 4 ? On y avait écrit :

```
private List<string> objets = new List<string>();
```

Cette **initialisation à la déclaration** est un raccourci. L'équivalent classique passe par le constructeur :

```
class Inventaire
{
    private List<string> objets;

    public Inventaire()
    {
        objets = new List<string>();
    }
}
```

Les deux formes font exactement la même chose. La forme avec constructeur est plus visible et plus flexible : tu peux y mettre **tout ce que tu veux** au moment de la création, pas seulement initialiser des attributs (lecture d'un fichier, log de création, ouverture d'une connexion…).

Partie 9 — Tout assembler

Voici la classe `Personnage` finale, avec encapsulation, méthodes et constructeurs :

```
class Personnage
{
    private int pv;

    public string Nom     { get; set; }
    public string Classe  { get; set; }
    public int    PvMax   { get; set; }
    public int    Attaque { get; set; }
    public int    Defense { get; set; }
    public int    Niveau  { get; set; }

    public int Pv
    {
        get => pv;
        set
        {
            if (value < 0)     value = 0;
            if (value > PvMax) value = PvMax;
            pv = value;
        }
    }

    public bool EstVivant => pv > 0;

    // --- Constructeur principal ---
    public Personnage(string nom, string classe, int pvMax,
                      int attaque = 10, int defense = 10, int niveau = 1)
    {
        Nom     = nom;
        Classe  = classe;
        PvMax   = pvMax;
        Pv      = pvMax;
        Attaque = attaque;
        Defense = defense;
        Niveau  = niveau;
    }

    public void Attaquer(Personnage cible)
    {
        int degats = Attaque - cible.Defense;
        if (degats < 0) degats = 0;
        cible.Pv -= degats;
        Console.WriteLine($"{Nom} attaque {cible.Nom} et inflige {degats} dégâts.");
    }

    public override string ToString()
```

```
    {
        string etat = EstVivant ? "vivant" : "mort";
        return $"{Nom} ({Classe}) - Niv.{Niveau} - {Pv}/{PvMax} PV - {etat}";
    }
}
```

Test final

```
Personnage aragorn = new Personnage("Aragorn", "Guerrier", 100, attaque: 15);
Personnage legolas = new Personnage("Legolas", "Archer",    80, attaque: 18, defense: 7);
Personnage paysan  = new Personnage("Otto",    "Paysan",    30);

Console.WriteLine(aragorn);
Console.WriteLine(legolas);
Console.WriteLine(paysan);

aragorn.Attaquer(legolas);
legolas.Attaquer(aragorn);

Console.WriteLine(aragorn);
Console.WriteLine(legolas);
```

Trois lignes pour créer trois personnages — au lieu de vingt-et-une. Et chaque personnage est **garanti** dans un état valide dès sa naissance.

🧪 Exercices

Exercice 1 — Constructeur pour Voiture

Reprends ta classe `Voiture`. Ajoute un constructeur :

```
public Voiture(string marque, string couleur, int nombreDePortes = 5)
```

Refactore tout `Program.cs` pour utiliser ce constructeur. Vérifie que la version `new Voiture()` (sans paramètres) ne fonctionne plus — c'est voulu.

Bonus : ajoute un deuxième constructeur `public Voiture(string marque) : this(marque, "Blanc")` pour les voitures dont on connaît juste la marque.

Exercice 2 — Compte bancaire

Reprends ta classe `CompteBancaire`. Ajoute :

- Un constructeur qui prend le titulaire et un solde initial (par défaut 0).
- Une **validation à la création** : si le solde initial est négatif, le ramener à 0 (en affichant un avertissement).

```
CompteBancaire c1 = new CompteBancaire("Marie Dupont", 1000);
CompteBancaire c2 = new CompteBancaire("Paul Martin");       // solde = 0
CompteBancaire c3 = new CompteBancaire("X", -500);            // solde = 0 + avertissement
```

▸ Exercice 3 — Chaînage de constructeurs

Pour la classe `LampeConnectee`, écris **trois constructeurs** chaînés avec `: this( ... )` :

- `LampeConnectee(string couleur, int intensite, bool allumee)` — le complet.
- `LampeConnectee(string couleur)` — délègue au précédent avec intensité 80 et allumée.
- `LampeConnectee()` — délègue au précédent avec couleur "Blanc".

Test :

```
LampeConnectee l1 = new LampeConnectee("Rouge", 50, true);
LampeConnectee l2 = new LampeConnectee("Bleu");
LampeConnectee l3 = new LampeConnectee();
```

▸ Exercice 4 — PNJ aléatoire

Ajoute à la classe `Personnage` une **méthode statique** `GenererPnj()` qui crée un personnage aléatoire :

- nom tiré d'une petite liste (Otto, Greta, Hans, Inge…) ;
- classe tirée d'une liste (Paysan, Marchand, Garde…) ;
- pvMax entre 20 et 60 ;
- attaque/defense entre 5 et 15.

Utilise la classe `Random` de C# (`Random rng = new Random();`).

```
Personnage pnj = Personnage.GenererPnj();
Console.WriteLine(pnj);
```

✍️ À retenir

- Un **constructeur** porte le nom de la classe et n'a pas de type de retour.
- Si tu n'en écris aucun, C# fournit un constructeur **par défaut sans paramètre**. Dès que tu en écris un, ce constructeur par défaut **disparaît**.
- Le `new Classe(args)` exécute le constructeur correspondant et **garantit l'état initial**.
- `this` lève l'ambiguïté quand un paramètre porte le même nom qu'un attribut.
- Une classe peut avoir plusieurs constructeurs (**surcharge**), chaînés avec `: this( ... )` pour éviter la duplication.
- Les **valeurs par défaut** des paramètres rendent un constructeur unique très flexible.
- Avec un bon constructeur, un objet **naît dans un état valide** dès le `new` — fini les sept lignes d'affectation.

Suite

À ce stade, tu maîtrises la classe en C# : attributs, visibilité, propriétés, méthodes, constructeurs. Tu peux modéliser n'importe quelle entité simple. Les prochains chapitres ouvriront la dimension *relationnelle* : comment des objets en contiennent d'autres (composition), comment une classe peut hériter d'une autre (héritage), et comment plusieurs classes différentes peuvent partager un même comportement (polymorphisme).



GESTION DE PROJETS

Introduction à la Gestion de Projet

La **gestion de projet** est l'ensemble des méthodes, outils et pratiques permettant de **planifier, organiser et piloter un projet** afin d'atteindre des objectifs précis dans un délai et avec des ressources définies. Un projet se distingue par son caractère **temporaire** (il a un début et une fin) et par sa **finalité unique** (il vise à produire un livrable spécifique, comme un logiciel, une application ou un service).

En informatique, la gestion de projet est essentielle pour **structurer le développement de logiciels, la mise en place d'infrastructures ou la conception de systèmes complexes**. Sans une gestion efficace, les projets risquent d'être mal organisés, de dépasser les délais ou le budget, ou encore de ne pas répondre aux attentes des utilisateurs. Que ce soit avec des approches **classiques** (en cascade) ou **agiles**, bien gérer un projet informatique permet d'optimiser le travail des équipes, d'anticiper les risques et d'assurer la réussite du produit final. 🚀

Le carré des contraintes

> *Au-delà du triangle d'or : le modèle à quatre sommets de Jurgen Appelo pour piloter un projet.*

Pourquoi ce cours ?

Dans n'importe quel projet en équipe, tu vas vivre la même tension : trop à faire, pas assez de temps, une équipe imparfaite, et un client (ou un prof, ou un public) qui veut quand même un résultat de qualité.

Ce cours te donne un **outil mental** pour rendre ces tensions visibles, en parler, et faire des choix lucides. Cet outil s'appelle le **carré des contraintes**, proposé par Jurgen Appelo dans *Management 3.0* (2010) en évolution du classique **triangle d'or**.

Principe fondamental

> ***Si un sommet varie dans un sens, soit les 2 sommets adjacents varient dans le même sens, soit le sommet opposé varie dans le sens inverse.***

C'est la règle unique qui régit toute la mécanique du modèle. Le reste du cours en dérive.

Objectifs d'apprentissage

À la fin de cette séquence, tu seras capable de :

1. **Identifier** les quatre sommets du carré des contraintes et donner leur équivalent en anglais et en français.
2. **Expliquer** le principe fondamental d'arbitrage : adjacents même sens / opposé sens inverse.
3. **Décomposer** la notion de ressources entre les outils et la capacité humaine (formule à quatre facteurs).
4. **Analyser** un projet en identifiant le sommet modifié et les sommets compensateurs.
5. **Argumenter** un choix d'arbitrage dans une réunion d'équipe agile en s'appuyant sur le carré.

Cinq notions-clés à retenir

#	NOTION	IDÉE CENTRALE
1	**Triangle d'or**	Le modèle classique : Scope, Time, Cost. Critiqué par Appelo car la qualité y est implicite.
2	**Carré des contraintes**	Quatre sommets explicites : **Scope, Time, Quality, Resources**.
3	**Adjacents vs opposé**	Si je modifie un sommet, soit les deux adjacents bougent dans le même sens, soit l'opposé bouge en sens inverse.
4	**Ressources = outils × humains**	Une équipe sans outils ne produit pas, des outils sans équipe non plus. La capacité humaine, elle, suit la formule `N × C × M × T` — multiplicative.
5	**Pas d'arbitrage gratuit**	Aucun projet ne maximise les quatre sommets en même temps. Le rôle du chef de projet est de rendre les arbitrages **explicites et négociés**.

---

1. Du triangle au carré

Le triangle d'or classique (Iron Triangle)

Depuis les années 1960, la gestion de projet se résume souvent à trois contraintes :

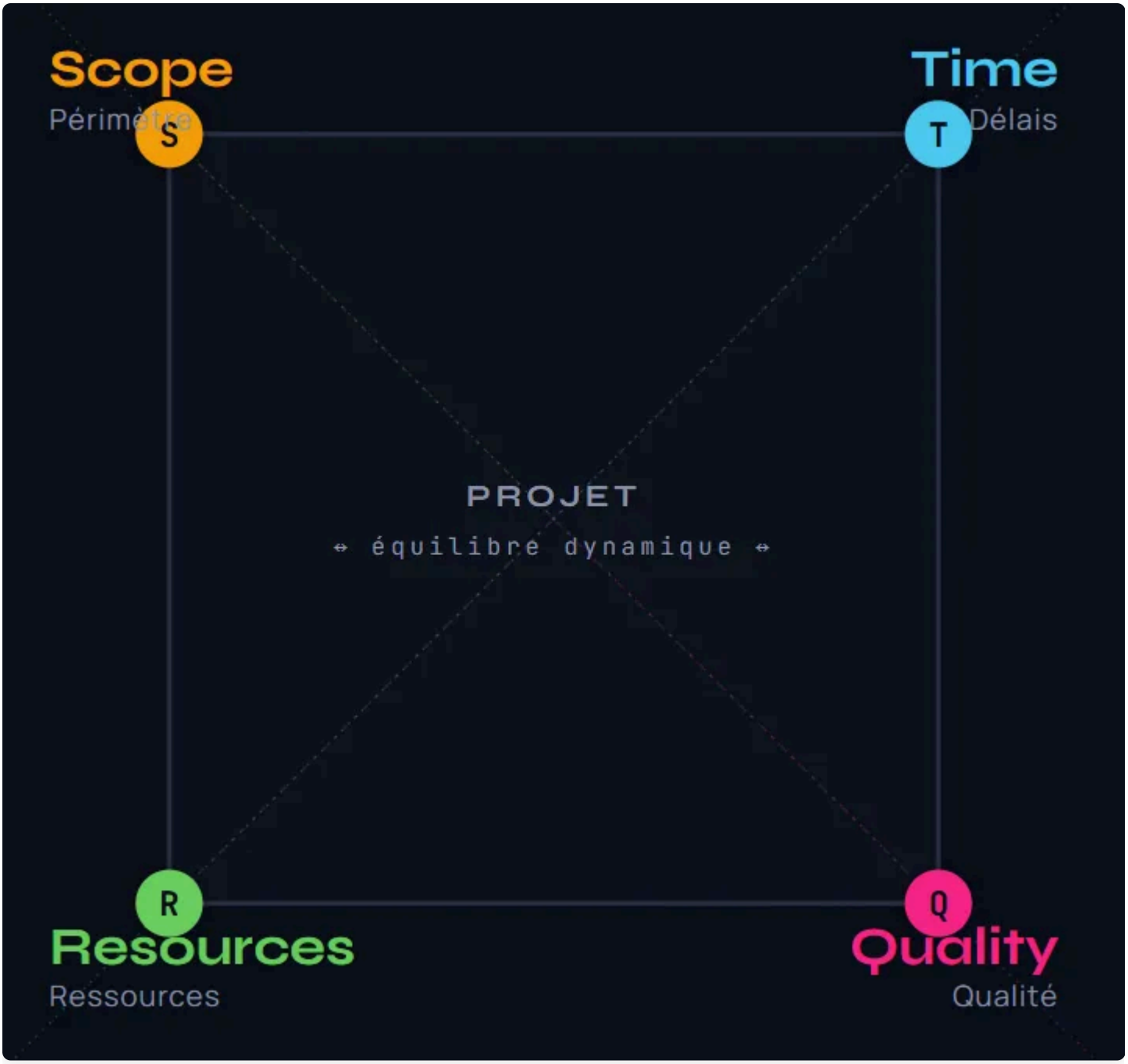
- **Scope** (périmètre) — ce qu'on doit livrer
- **Time** (délai) — combien de temps on a
- **Cost** (coût) — combien ça coûte

> *« Vite, pas cher, beaucoup de fonctionnalités — choisis-en deux. »*

C'est élégant, mais Appelo trouve que **la qualité y est traitée comme un sous-produit silencieux**. Elle se dégrade quand on serre les trois autres, mais personne n'en parle ouvertement.

Le carré d'Appelo

Appelo propose d'ajouter explicitement la **Qualité** comme quatrième sommet :



Voisinages :

SOMMET	ADJACENTS (CÔTÉ DU CARRÉ)	OPPOSÉ (DIAGONALE)
Scope	Time, Resources	Quality
Time	Scope, Quality	Resources
Quality	Time, Resources	Scope
Resources	Scope, Quality	Time

Cette topologie est essentielle : c'est elle qui détermine *qui* doit bouger *quand* tu modifies un sommet.

2. Les quatre sommets en détail

↖ Sommet haut-gauche : Scope *(Périmètre)*

Définition. L'ensemble des fonctionnalités, livrables et exigences qui définissent **ce que** le projet doit produire. C'est la *frontière* du projet : ce qui est inclus, ce qui est exclu.

Vocabulaire.

- GB *Scope, requirements, features, deliverables*
- FR Périmètre, exigences, fonctionnalités, livrables

Exemple. Un site de réservation avec catalogue, calendrier, paiement en ligne, comptes utilisateurs, scan QR et back-office → **scope large**. Le même site avec uniquement le catalogue et le calendrier → **scope réduit**.

Piège classique. Le **scope creep** : le périmètre grossit en douce au fil du projet (« on pourrait juste ajouter… »). Sans arbitrage, c'est le délai, la qualité ou les ressources qui paient l'addition.

↗ Sommet haut-droite : Time *(Délais)*

Définition. La durée totale disponible pour livrer le projet. Inclut la **date de fin** (deadline), les **jalons** intermédiaires, et le **rythme** de travail.

Vocabulaire.

- GB *Time, schedule, deadline, milestones*
- FR Délais, temps imparti, échéance, jalons

Exemple. Une soirée d'entreprise est prévue le 14 février. Cette date **ne bouge pas**. Si le client demande une nouvelle catégorie de questions trois semaines avant, c'est le scope ou la qualité qui devra s'ajuster — jamais le délai.

Particularité. Le temps est souvent **la contrainte la plus rigide** : un salon, une rentrée scolaire, une obligation légale, un lancement marketing — ces dates sont externes au projet et ne se négocient pas.

↘ Sommet bas-droite : Quality *(Qualité)*

Définition. Le niveau d'exigence du livrable : robustesse du code, ergonomie, accessibilité, performance, sécurité, design, absence de bugs, conformité aux normes.

Vocabulaire.

- GB *Quality, robustness, usability, performance*
- FR Qualité, robustesse, ergonomie, performance

Exemple. Une app médicale doit être validée **sans bug critique** — la qualité est imposée par la loi. Un prototype présenté en interne peut tolérer des bugs mineurs : on ajuste après.

Le piège de la qualité sacrifiée. Quand on dit *« on livrera vite, on corrigera après »*, on crée de la **dette technique** : du code mal écrit qu'il faudra payer (avec intérêts) plus tard. Appelo insiste : rendre la qualité explicite, c'est la rendre négociable **en conscience**, pas par défaut.

↙ Sommet bas-gauche : Resources *(Ressources)*

Définition. Tout ce qui est mobilisé pour réaliser le projet : équipe humaine, outils, matériel, budget, locaux.

Vocabulaire.

- GB *Resources, people, tools, capacity, budget*
- FR Ressources, équipe, outils, capacité, budget

C'est le sommet **le plus mal compris** — et c'est sur lui qu'on va passer un peu plus de temps.

---

3. Zoom — Décomposer les ressources

> ⚠️ ***Note importante.*** *Ce qui suit est un* ***enrichissement personnel*** *du modèle d'Appelo,* ***pas la version d'origine****. Dans Management 3.0, Appelo traite les ressources de manière assez globale — équipe et budget réunis. La décomposition ci-dessous est une grille de lecture plus fine pour mieux piloter ce qu'il y a derrière la simple étiquette « ressources ».*

Deux grandes familles

Pour un projet, les ressources se regroupent en deux grandes familles :

1. **Les outils** (`O`) — tout ce qui n'est pas humain mais qui permet de produire : matériel (ordinateurs, serveurs, hardware spécialisé), logiciels (IDE, suites de design, outils de build), infrastructure (cloud, CI/CD, bases de données), locaux, licences, accès aux specs et à la documentation, budget.
2. **La capacité humaine** (`H`) — ce que l'équipe peut réellement produire, qui dépend de quatre facteurs détaillés ci-dessous.

Les deux dimensions se combinent de manière **multiplicative** :

> $R = O \times H$

Pourquoi multiplicatif ? Parce qu'une équipe top sans serveur, sans IDE, sans accès aux specs → bloquée. Et des outils flambant neufs sans personne pour les utiliser → rien ne sort. Si l'une des deux dimensions tend vers zéro, l'ensemble tend vers zéro.

La capacité humaine : quatre facteurs

> $H = N \times C \times M \times T$

Où :

- **N — Nombre de personnes** dans l'équipe (de 1 à n)
- **C — Compétence** : ce qu'elles savent faire dans le domaine du projet
- **M — Motivation** : leur envie réelle d'avancer sur *ce* projet
- **T — Temps disponible** : la part de leur temps de travail qu'elles peuvent vraiment y consacrer

⚠️ Une remarque essentielle sur les pourcentages

Dans les exemples qui suivent, on utilise des pourcentages (« compétence à 95 % », « motivation à 0 % »…). **Ces chiffres ne sont pas des mesures absolues.** Personne ne calcule scientifiquement « 78 % de motivation » ou « 42 % de compétence » — c'est impossible.

Les pourcentages sont là **uniquement pour faciliter la compréhension** : ils permettent de visualiser intuitivement les ordres de grandeur et l'effet multiplicatif. En pratique, trois niveaux suffisent souvent — *élevé / moyen / faible* — voire juste la question *« est-ce qu'il manque quelque chose ? »*.

L'important n'est **pas la précision du chiffre**. C'est de reconnaître que **tous les facteurs jouent en même temps**, et que **tout facteur très bas fait chuter l'ensemble**.

Pourquoi c'est *multiplicatif* et pas *additif*

C'est la clé du modèle. Une multiplication a une propriété particulière : **si un seul facteur est à zéro, le résultat est zéro**, peu importe les autres.

Regardons quatre cas pour s'en convaincre :

CAS 1 — LE DÉVELOPPEUR EXPERT MAIS DÉMOTIVÉ

> *1 personne · compétence très élevée ·* ***motivation nulle*** *· temps total disponible →* `H ≈ 1 × 0,95 × 0 × 1 = 0`

Le projet n'avance pas. Aucune productivité. La compétence sans motivation = zéro. C'est **la grande leçon d'Appelo** : la motivation n'est pas un « bonus » qu'on ajoute, c'est un **facteur multiplicatif**. Sans elle, tout le reste est neutralisé.

CAS 2 — L'ÉQUIPE MOTIVÉE MAIS SANS COMPÉTENCE

> *5 personnes ·* ***compétence très faible*** *· motivation maximale · temps total disponible →* `H ≈ 5 × 0,05 × 1 × 1 = 0,25`

L'enthousiasme ne remplace pas le savoir-faire. Une équipe ultra-motivée mais qui découvre une techno la veille du sprint ne livrera pas une plateforme robuste.

CAS 3 — L'EXPERT SANS TEMPS DISPONIBLE

> *1 personne · compétence maximale · motivation maximale ·* ***temps disponible très faible*** *(autres projets prioritaires) →* `H ≈ 1 × 1 × 1 × 0,05 = 0,05`

Un excellent profil à quelques % de son temps fournit la même chose qu'un débutant à quelques % : presque rien. Le temps disponible est crucial — et souvent surestimé.

CAS 4 — L'ÉQUIPE ÉQUILIBRÉE

> *5 personnes · compétence assez bonne · motivation bonne · temps partiel →* `H ≈ 5 × 0,7 × 0,8 × 0,6 ≈ 1,7`

Soit environ **17 %** d'un maximum théorique de 10 personnes à plein régime sur tous les facteurs. C'est une équipe **correcte et réaliste** : on ne travaille jamais à fond sur tous les axes en même temps.

Comment évaluer ta propre équipe

Avant chaque sprint, pose-toi (en équipe) les quatre questions :

1. **N —** Sommes-nous combien à vraiment travailler dessus ? (Distinguer les présents des actifs.)
2. **C —** Maîtrisons-nous les technos nécessaires, ou faut-il apprendre en chemin ?
3. **M —** Est-ce que tout le monde *a envie* d'avancer ? Y a-t-il un sujet bloquant ?
4. **T —** Combien d'heures réelles peut-on y consacrer cette semaine ?

Et n'oublie pas la première moitié de la formule : **a-t-on les outils nécessaires ?** Sans accès au serveur de test, sans la bonne licence logiciel, sans la doc à jour, même la meilleure équipe sera bloquée.

Si l'un des facteurs (outils inclus) est très bas → c'est lui qu'il faut adresser **en priorité** avant d'ajouter du scope.

---

4. La mécanique d'arbitrage

C'est le **cœur du modèle**. Pour chaque situation, on applique le principe fondamental : **soit** les deux adjacents bougent dans le même sens (🟢 vert), **soit** l'opposé bouge en sens inverse (🔴 rouge).

Situation A — Le périmètre augmente (Scope ↑)

> *Un client ajoute une fonctionnalité en cours de projet.*

🟢 **Soit les deux adjacents augmentent :**

- ↑ **Time** — il faut plus de temps
- ↑ **Resources** — il faut plus de ressources

🔴 **Soit l'opposé diminue :**

- ↓ **Quality** — la qualité baisse en compensation

Situation B — Le délai se raccourcit (Time ↓)

> *La direction avance la deadline de trois semaines.*

🟢 **Soit les deux adjacents diminuent :**

- ↓ **Scope** — on réduit les fonctionnalités
- ↓ **Quality** — on accepte une qualité moindre

🔴 **Soit l'opposé augmente :**

- ↑ **Resources** — on ajoute des renforts

▸ Situation C — Les ressources diminuent (Resources ↓)

> *Un développeur quitte l'équipe à mi-projet.*

🟢 **Soit les deux adjacents diminuent :**

- ↓ **Scope** — on coupe des features
- ↓ **Quality** — la qualité chute

🔴 **Soit l'opposé augmente :**

- ↑ **Time** — on allonge le délai

▸ Situation D — La qualité doit monter (Quality ↑)

> *Audit qualité imposé, ou cible plus exigeante.*

🟢 **Soit les deux adjacents augmentent :**

- ↑ **Time** — il faut plus de temps pour peaufiner
- ↑ **Resources** — plus de relecture, plus de tests

🔴 **Soit l'opposé diminue :**

- ↓ **Scope** — on livre moins, mais mieux

▸ Le rôle du chef d'équipe

À chaque demande de changement, **trois questions à se poser à voix haute** :

1. Quel sommet veut bouger, et dans quel sens ?
2. Qui sont les adjacents, qui est l'opposé ?
3. Quel scénario de compensation choisit-on : **adjacents même sens** ou **opposé sens inverse** ?

L'erreur classique : *« on garde tout pareil et on encaisse ».* Ça n'existe pas. Le carré rend l'arbitrage **explicite** au lieu de le subir en silence.

---

5. Lien avec l'écoresponsabilité numérique

Un sommet souvent oublié mais qui devrait être discuté : **toute fonctionnalité ajoutée a un coût environnemental**.

- Plus de scope → plus de code, plus de serveurs, plus de stockage, plus de bande passante consommée.
- Plus de ressources humaines → plus de déplacements, de hardware, de chauffage de bureaux.
- Plus de qualité (notamment perf) → souvent moins de gaspillage énergétique côté utilisateur.

Quand tu négocies le scope en équipe, pose la question : **cette fonctionnalité justifie-t-elle son empreinte ?** Le carré devient alors aussi un outil d'arbitrage éthique. C'est la posture **Lean & Green** de la gestion de projet moderne.

6. Exercices

Exercice 1 — Application du principe

Pour chaque scénario, indique les **deux scénarios de compensation possibles** selon le principe fondamental :

1. La qualité doit augmenter (audit imposé).
2. Le scope doit diminuer (le client coupe une feature).
3. Les ressources doivent augmenter (recrutement d'un dev).
4. Le délai s'allonge (livraison repoussée).

Réponse attendue par scénario :

- 🟢 **Adjacents :** quels sommets et dans quel sens ?
- 🔴 **Opposé :** quel sommet et dans quel sens ?

Exercice 2 — Diagnostic d'équipe

Évalue la capacité réelle de ton équipe agile pour la semaine prochaine. Pour chaque facteur, contente-toi d'un niveau *élevé / moyen / faible* (pas besoin de pourcentages précis) :

- **Outils** disponibles (matériel, logiciels, accès aux specs)
- **N** — nombre de personnes vraiment actives
- **C** — compétence collective sur les technos requises
- **M** — motivation
- **T** — temps réellement disponible

→ Identifie le **maillon faible** (le facteur le plus bas) et propose **une action concrète** pour le faire remonter.

Exercice 3 — Négociation

Le client demande une fonctionnalité supplémentaire à 10 jours de la livraison. Rédige un message Slack ou e-mail expliquant **les deux options possibles** (adjacents même sens / opposé sens inverse) et **ta recommandation argumentée**.

Exercice 4 — Étude de cas

Choisis un projet réel que tu connais (un projet open source, le développement d'un jeu vidéo connu, un grand chantier public…). Analyse-le à travers le carré :

- Quel sommet a été **préservé à tout prix** ?
- Quel sommet a été **sacrifié** ?
- Quel scénario d'arbitrage a été appliqué (adjacents même sens, ou opposé sens inverse) ?

Pour aller plus loin

- **Jurgen Appelo**, *Management 3.0 — Leading Agile Developers, Developing Agile Leaders*, Addison-Wesley, 2010 (chap. 1 & 4).
- **Jurgen Appelo**, *Managing for Happiness*, Wiley, 2016 (sur la motivation comme facteur multiplicatif).
- **The Mythical Man-Month**, Fred Brooks, 1975 (la fameuse loi : *« ajouter des développeurs à un projet en retard le retarde davantage »* — illustration du modèle multiplicatif).
- **The Phoenix Project**, Gene Kim et al. (sur la dette technique et l'arbitrage qualité dans un contexte DevOps).

« Le triangle vous dit ce que vous pouvez optimiser. Le carré vous rappelle ce que vous sacrifiez. »

Méthodes agiles

> *L'**Agilité** est une approche moderne de gestion de projet qui repose sur la flexibilité, l'adaptabilité et la collaboration. Contrairement aux méthodes traditionnelles, qui suivent un plan rigide du début à la fin, les méthodes **Agiles** permettent d'ajuster le projet en cours de route en fonction des retours et des priorités changeantes.*

Qu'est-ce que l'Agilité en gestion de projet ?

L'**Agilité** est une approche moderne de gestion de projet qui repose sur la flexibilité, l'adaptabilité et la collaboration. Contrairement aux méthodes traditionnelles, qui suivent un plan rigide du début à la fin, les méthodes **Agiles** permettent d'ajuster le projet en cours de route en fonction des retours et des priorités changeantes.

L'Agilité est particulièrement utilisée dans le développement logiciel, mais elle s'étend aujourd'hui à d'autres domaines comme le marketing, la gestion de produit ou encore l'industrie.

Les principes des méthodes Agiles

Les méthodes Agiles sont basées sur le **Manifeste Agile** (2001), qui repose sur **quatre valeurs fondamentales** :

1. **Les individus et leurs interactions** plus que les processus et les outils.
2. **Un logiciel fonctionnel** plus qu'une documentation exhaustive.
3. **La collaboration avec les clients** plus que la négociation contractuelle.
4. **L'adaptation au changement** plus que le suivi d'un plan rigide.

L'Agilité encourage donc une approche plus dynamique et humaine du travail, axée sur la satisfaction du client et l'amélioration continue.

Les principales méthodes Agiles

Plusieurs méthodologies appliquent les principes de l'Agilité. Voici les plus connues :

1 Scrum

- Fonctionne en **sprints** (cycles courts de 1 à 4 semaines).
- Rôles clés : **Scrum Master** (facilitateur), **Product Owner** (vision du produit) et **équipe de développement**.
- Utilise un **tableau Scrum** pour suivre les tâches et une **réunion quotidienne** (Daily Scrum) pour ajuster le travail.

2 Kanban

- Utilise un **tableau Kanban** pour visualiser l'avancement des tâches.

- Favorise un **flux de travail continu**, en limitant le nombre de tâches en cours (**WIP – Work In Progress**).
- Permet une grande flexibilité, car il n'impose pas de cycles rigides.

3️⃣ Extreme Programming (XP)

- Axé sur les **bonnes pratiques du développement logiciel** (tests automatisés, programmation en binôme, intégration continue).
- Vise à améliorer la qualité du code et la réactivité aux changements.

4️⃣ Lean

- Inspiré du **système de production de Toyota**.
- Objectif : **réduire les gaspillages** et optimiser la valeur apportée au client.
- Met l'accent sur l'amélioration continue et la simplification des processus.

Pourquoi utiliser une méthode Agile ?

✅ **Plus de flexibilité** : on peut adapter le projet aux évolutions et aux imprévus.
✅ **Une meilleure collaboration** : les équipes travaillent ensemble et avec les clients.
✅ **Livraison plus rapide** : des petites versions du projet sont livrées régulièrement.
✅ **Amélioration continue** : les erreurs sont corrigées rapidement et les processus sont optimisés au fur et à mesure.

Les méthodes Agiles permettent donc de **répondre plus efficacement aux besoins des clients** tout en favorisant un travail d'équipe plus fluide et motivant. 🚀

Les User Stories

> *Une user story est une description simple, en langage naturel courant, d'un besoin ou d'une attente d'un utilisateur ou d'un client, dans le cadre d'un projet logiciel ou d'un produit.*

Pourquoi les user stories existent

As-tu déjà construit une fonctionnalité pour te rendre compte, trop tard, que **ce n'est pas ce que l'utilisateur voulait** ? C'est précisément ce problème que les **user stories** cherchent à résoudre.

Les user stories sont :

- **petites**
- **simples**
- **puissantes**
- et surtout **centrées sur l'utilisateur**

Elles permettent aux équipes de rester concentrées sur **la valeur réelle pour l'utilisateur**, et non sur ce que l'équipe *suppose* être utile.

Elles fonctionnent :

- en Agile,
- en méthodologie hybride,
- et même dans des projets encore peu structurés.

Définition d'une user story

Une **user story** est :

> *une **courte description**, en langage simple, d'une **fonctionnalité ou capacité**, racontée **du point de vue de l'utilisateur**.*

Son objectif principal est clair :

👉 **s'assurer que ce que l'on construit répond réellement aux besoins des utilisateurs**.

Ce qu'une user story n'est PAS

- ❌ ce n'est pas une spécification complète
- ❌ ce n'est pas une description technique
- ❌ ce n'est pas une liste détaillée de règles

✅ Une user story est **un point de départ pour une discussion**, pas une documentation figée.

Le problème des exigences (requirements)

Dans tout projet, une question revient sans cesse :

> *Comment s'assurer que les personnes qui construisent le produit comprennent réellement ce que les utilisateurs veulent ?*

Les user stories apportent une réponse simple :

- elles favorisent une **communication humaine, claire et centrée sur les besoins**
- elles évitent un "dump" de détails techniques incompréhensibles
- elles **font le lien** entre :
  - ceux qui **ont un besoin**
  - et ceux qui **construisent la solution**

👉 Elles comblent le fossé entre métier et technique.

Le format classique : WHO – WHAT – WHY

Les user stories suivent un format volontairement simple :

```
As a [WHO]
I want / I can [WHAT]
So that [WHY]
```

En français :

> ***En tant que*** *[qui]* ***Je veux / je peux*** *[quoi]* ***Afin de*** *[pourquoi]*

Rôle de chaque partie

- **WHO (Qui)** L'utilisateur ou le rôle concerné → "utilisateur enregistré", "administrateur", "nouvel utilisateur", etc.
- **WHAT (Quoi)** Ce que l'utilisateur veut pouvoir faire → une action, une capacité, un besoin
- **WHY (Pourquoi)** La valeur, le bénéfice, la raison → ce que cela apporte à l'utilisateur

Règle fondamentale

🚫 **Pas de HOW (comment)**

On **ne décrit pas la solution technique**. On décrit **le besoin et la valeur**, pas l'implémentation.

Qui écrit les user stories ?

Réponse courte :

👉 **le Product Owner**

Pourquoi ?

- il est responsable du **backlog**
- il garantit que chaque story est :
  - claire
  - utile
  - alignée avec la vision produit

Réponse complète (la plus importante)

✅ **Les bonnes user stories sont un travail d'équipe**

- n'importe qui peut proposer une story
- développeurs, designers, testeurs doivent participer
- les meilleures stories émergent souvent :
  - d'une discussion
  - d'un tableau blanc
  - d'un "Et si l'utilisateur faisait plutôt ça ?"

👉 Une user story devient vraiment utile **quand elle est enrichie collectivement**.

Les 3 C's des user stories (Ron Jeffries)

Pour bien comprendre ce qu'est *vraiment* une user story, Ron Jeffries a défini les **3 C's** :

1️⃣ Card (la carte)

À l'origine, les user stories étaient écrites sur des **cartes physiques**.

Pourquoi ?

- elles sont **simples**
- manipulables
- regroupables
- jetables

Elles donnent un message important :

👉 **Une user story n'est pas un document permanent**

Même aujourd'hui, dans les outils numériques :

- une story **n'est pas une spécification figée**
- **elle n'est pas le requirement**
- elle est seulement **un pointeur vers le besoin**

▸ 2️⃣ Conversation (la conversation)

C'est **le cœur du processus**.

Chaque user story est :

> ***une promesse d'avoir une conversation avant de développer***

Questions typiques :

- "Peux-tu nous en dire plus ?"
- "À quoi ressemble le succès ?"
- "Et si l'utilisateur fait X ?"
- "Que se passe-t-il dans ce cas ?"

⚠️ Sans cette conversation :

- le Product Owner est tenté d'écrire des specs énormes
- on perd tout l'intérêt des stories

Important :

> *Le fait que le PO "veuille quelque chose" ne signifie pas que l'équipe doit l'implémenter tel quel.*

L'équipe doit :

- questionner
- challenger
- proposer d'autres solutions

👉 **La conversation fait partie intégrante de la story.**

▸ 3️⃣ Confirmation (la confirmation)

La confirmation correspond aux **critères d'acceptation**.

Ils répondent à la question :

> *Comment saura-t-on que la story est terminée, et bien terminée ?*

Exemples :

- "La story doit faire ceci"
- "Elle ne doit pas faire cela"
- "Si X arrive, alors Y doit se produire"

✅ Ces critères :

- émergent naturellement des conversations
- donnent de la clarté
- évitent les malentendus

Exemples de user stories (connexion à un site)

Story de base

> *En tant qu'utilisateur enregistré, je dois me connecter afin d'accéder au système.*

Autres points de vue

- **Mot de passe oublié**

  > *En tant qu'utilisateur distrait, je peux demander un rappel de mot de passe afin de pouvoir me connecter même si je l'ai oublié.*

- **Nouvel utilisateur**

  > *En tant que nouvel utilisateur, je peux créer un compte afin de me connecter et enregistrer mes préférences.*

- **Utilisateur fidèle (expérience positive)**

  > *En tant qu'utilisateur revenant pour la énième fois, je reçois une récompense afin de me sentir reconnu et encouragé à continuer.*

👉 Remarque : La story dit *"nième fois"* sans préciser 3, 5 ou 20. Ce détail sera décidé **plus tard**, au moment de l'implémentation.

Où sont les détails si la story est si courte ?

Une user story est souvent **une seule phrase**.

C'est normal.

- les détails viennent :
  - pendant les conversations
  - via les critères d'acceptation
  - au moment où la story est prête à être développée

✅ La brièveté est une force, pas une faiblesse.

À retenir (synthèse)

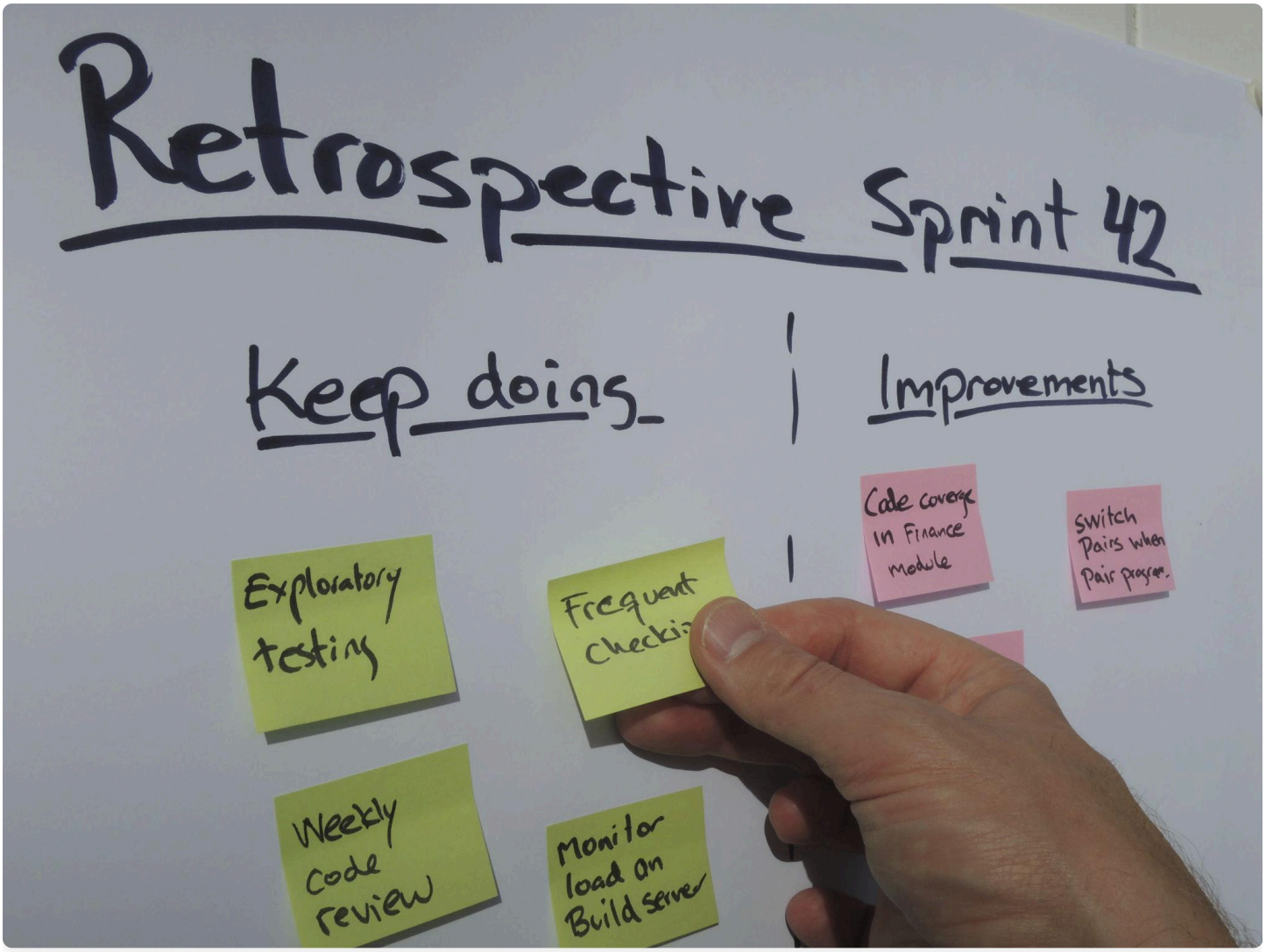
- Une user story est **courte, simple, centrée utilisateur**
- Elle décrit **WHO – WHAT – WHY**, jamais le HOW
- Elle est :
  - un **pointeur vers un besoin**
  - une **promesse de conversation**
  - validée par des **critères de confirmation**
- Les 3 C's sont fondamentaux :
  - **Card**
  - **Conversation**
  - **Confirmation**
- Les user stories sont un **outil de collaboration**, pas juste de documentation

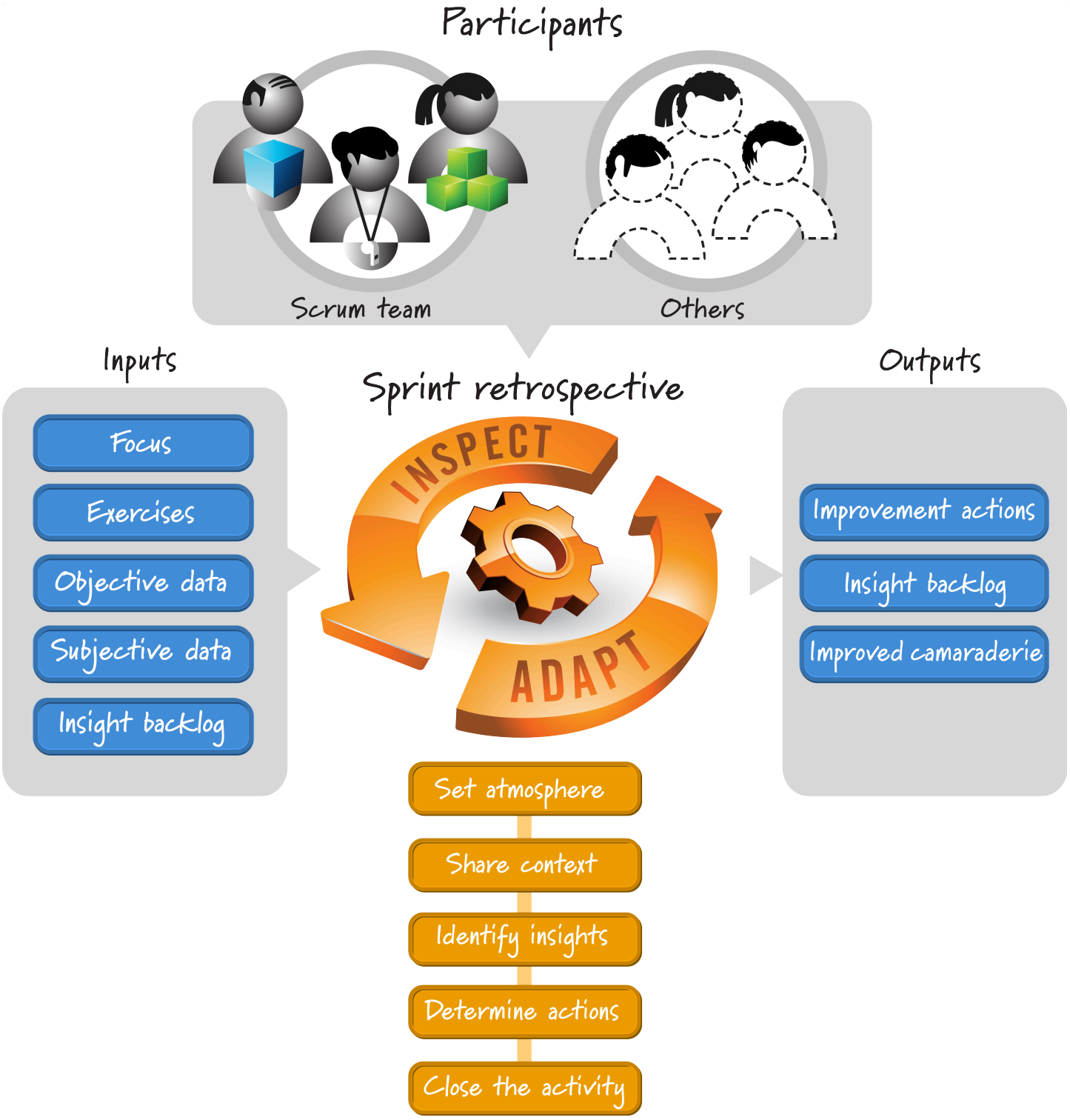
---

A rétrospective

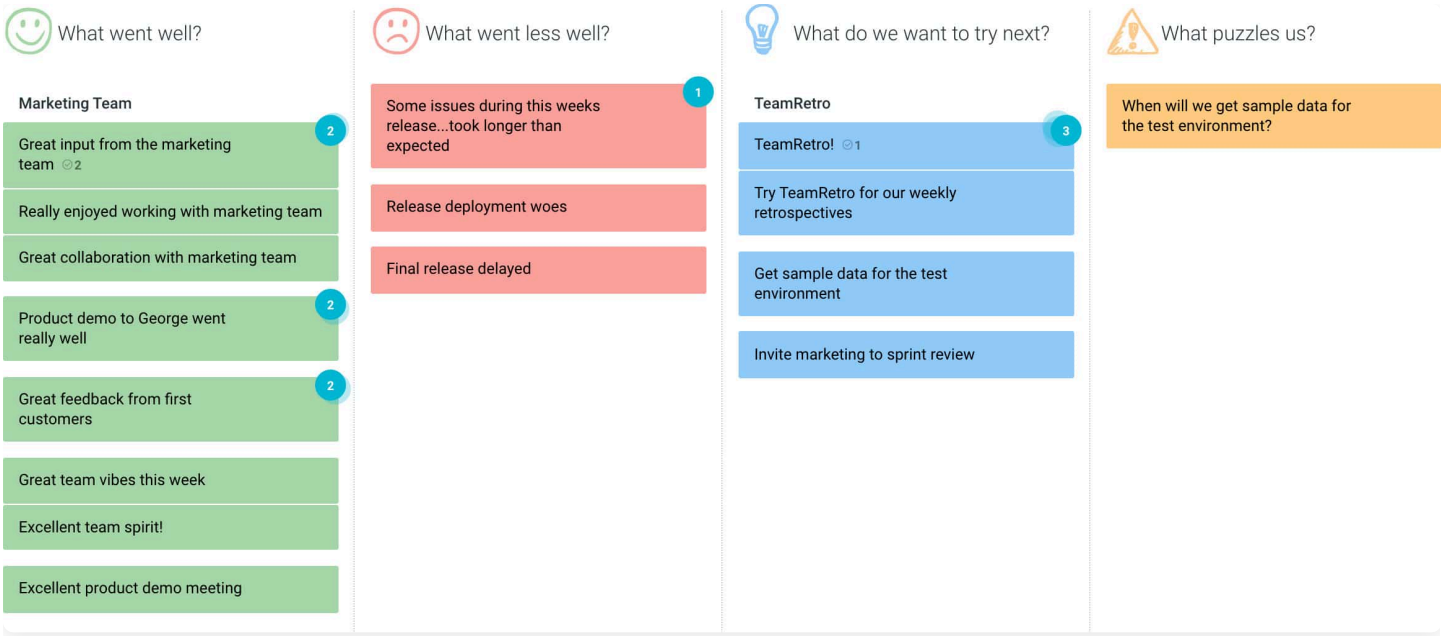
> *Une user story est une description simple, en langage naturel courant, d'un besoin ou d'une attente d'un utilisateur ou d'un client, dans le cadre d'un projet logiciel ou d'un produit.*




Retrospective Sprint 42
Keep doing
Improvements
Exploratory testing
Weekly code review
Monitor load on Build server
Code coverage in Finance module
switch Pairs when pair progra



Copyright © 2012, Kenneth S. Rubin and Innolution, LLC. All Rights Reserved.



La rétrospective agile en Scrum

La rétrospective est une cérémonie essentielle de Scrum : elle permet à l'équipe de **prendre du recul**, d'analyser sa façon de travailler et de **s'améliorer en continu** après chaque sprint.

Objectifs de la leçon

À la fin de cette leçon, tu dois être capable de :

- expliquer ce qu'est une rétrospective Scrum,
- comprendre son objectif réel (au-delà du simple "débrief"),
- identifier les étapes clés d'une rétrospective,
- proposer des actions d'amélioration concrètes pour un sprint suivant.

Qu'est-ce qu'une rétrospective Scrum ?

La **rétrospective** est une réunion organisée **à la fin d'un sprint**, après la revue de sprint. Toute l'équipe Scrum y participe : développeurs, Scrum Master et Product Owner.

Son but n'est **pas** de juger les personnes, mais d'analyser **le processus de travail** :

- ce qui a bien fonctionné,
- ce qui a posé problème,
- ce qui peut être amélioré dès le prochain sprint.

À quoi sert vraiment la rétrospective ?

La rétrospective sert à **s'améliorer en continu**. Sans elle, l'équipe répète souvent les mêmes erreurs.

Concrètement, elle permet de :

- améliorer l'organisation de l'équipe,
- fluidifier la communication,
- identifier des blocages récurrents,
- ajuster les méthodes de travail (outils, règles, rythme…).

👉 **Une équipe agile progresse sprint après sprint grâce à la rétrospective.**

Comment se déroule une rétrospective ?

Une rétrospective suit généralement une structure simple.

Temps de parole sécurisé Chacun peut s'exprimer librement, sans accusation personnelle.

Analyse du sprint écoulé On répond à des questions comme :

- Qu'est-ce qui a bien fonctionné ?
- Qu'est-ce qui a moins bien fonctionné ?
- Qu'est-ce qui nous a bloqués ?

Recherche d'améliorations L'équipe propose des idées concrètes pour faire mieux.

Décision d'actions On choisit **1 à 3 actions maximum**, réalistes et mesurables, à appliquer au sprint suivant.

Exemple simple de rétrospective

Méthode **Start / Stop / Continue** :

- **Start** : ce qu'on devrait commencer à faire *Exemple : faire un daily stand-up plus court et structuré.*
- **Stop** : ce qu'on devrait arrêter *Exemple : travailler sans mettre à jour le tableau Scrum.*
- **Continue** : ce qui fonctionne bien *Exemple : bien découper les tâches avant le sprint.*

Règles importantes à respecter

- On critique **les pratiques**, pas les personnes.
- Tout le monde participe.
- Les actions décidées doivent être testées **au sprint suivant**.
- Une rétrospective sans action concrète ne sert à rien.

À retenir

- La rétrospective a lieu **à la fin de chaque sprint**.
- Elle vise l'**amélioration continue** de l'équipe.
- Elle analyse le **processus**, pas les individus.
- Elle débouche sur **quelques actions concrètes**.
- Elle est indispensable au fonctionnement de Scrum.

Notes importantes

Positionnement pédagogique Cette leçon s'intègre naturellement dans un projet collaboratif (développement web, application, jeu, projet technique).

Erreur fréquente des élèves Les membres de l'équipe confondent souvent rétrospective et revue de sprint. 👉 Insister :

- *Revue* = on montre le produit
- *Rétrospective* = on améliore la manière de travailler

Lien avec les compétences transversales

- communication,
- esprit critique,
- collaboration,
- auto-évaluation.

La méthode Kanban



Qu'est-ce que la méthode Kanban ?

La méthode **Kanban** est une approche de gestion du travail qui permet de visualiser, organiser et optimiser les flux de travail. Elle est particulièrement utilisée dans le domaine du développement logiciel, de la gestion de projet et même dans la gestion de tâches personnelles.

Kanban repose sur l'utilisation d'un **tableau Kanban**, qui est un outil visuel permettant de suivre l'avancement des tâches. L'objectif est d'améliorer l'efficacité et la fluidité du travail en identifiant rapidement les blocages et en limitant le travail en cours (**WIP – Work In Progress**).

Le principe de la méthode Kanban

Kanban repose sur quelques principes clés :

1. **Visualisation du travail** : chaque tâche est représentée sous forme de carte (ou post-it) placée sur un tableau, qui est divisé en plusieurs colonnes représentant les différentes étapes du processus.
2. **Limiter le travail en cours (WIP – Work In Progress)** : il ne faut pas surcharger une étape avec trop de tâches pour éviter les goulots d'étranglement.
3. **Gérer le flux** : l'objectif est de maintenir un flux de travail fluide et d'éviter les blocages.
4. **Rendre explicites les règles du processus** : chaque équipe doit définir clairement ses règles pour garantir une bonne organisation.

5. **Amélioration continue** : Kanban est une méthode évolutive, qui s'adapte aux besoins en identifiant et en corrigeant les inefficacités.

L'objectif de la méthode Kanban

L'objectif principal de Kanban est **d'améliorer la productivité et l'efficacité en rendant le travail plus fluide et transparent**. En utilisant Kanban, une équipe peut :

- Réduire le temps nécessaire pour accomplir une tâche.
- Identifier rapidement les blocages.
- Adapter le processus de travail aux besoins réels.
- Travailler de manière plus collaborative et efficace.

Comment mettre en œuvre Kanban de façon simple ?

1. Créer un tableau Kanban

Le tableau Kanban peut être **physique** (un tableau blanc avec des post-it) ou **numérique** (via des outils comme Trello, Jira, ou Notion). Il est divisé en colonnes qui représentent les différentes étapes du processus de travail.

Un exemple classique de tableau Kanban :

À FAIRE (TO DO)	EN COURS (IN PROGRESS)	TERMINÉ (DONE)
Rédiger le rapport	Corriger un bug	Publier l'article
Créer une maquette	Développer une fonctionnalité	Valider le projet

---

2. Ajouter les tâches sous forme de cartes

Chaque tâche est représentée sous forme d'une **carte** (ou post-it). Une carte contient généralement :

- Un titre clair de la tâche.
- Une description si nécessaire.
- Une personne assignée.
- Une date limite (optionnelle).

---

3. Déplacer les tâches au fur et à mesure de leur avancement

Les tâches avancent progressivement à travers les colonnes du tableau en fonction de leur état d'avancement. Par exemple :

1. Une nouvelle tâche commence dans la colonne **"À faire"**.
2. Lorsqu'une personne commence à travailler dessus, elle la déplace dans **"En cours"**.

3. Une fois la tâche terminée, elle est placée dans "**Terminé**".

▸ 4. Limiter le travail en cours (WIP)

Une règle importante de Kanban est de **limiter le nombre de tâches en cours**. Par exemple, si une équipe fixe une limite de 3 tâches en cours par personne, cela évite la surcharge et permet de mieux se concentrer sur les priorités.

▸ 5. Améliorer continuellement le processus

Kanban encourage une réflexion continue sur l'amélioration du flux de travail. Quelques questions à se poser régulièrement :

- Y a-t-il des blocages fréquents dans une étape ?
- Une colonne est-elle souvent surchargée ?
- Pouvons-nous mieux répartir les tâches ?
- Le tableau reflète-t-il bien notre manière de travailler ?

En ajustant les règles et les limites, l'équipe optimise progressivement son travail.

Conclusion : ce qu'il faut retenir

✅ **Kanban est une méthode visuelle de gestion du travail** qui repose sur un tableau avec des tâches organisées en colonnes.
✅ Son objectif est **d'optimiser le flux de travail et d'améliorer l'efficacité** en limitant le travail en cours.
✅ Il suffit de **créer un tableau**, d'y ajouter des tâches sous forme de cartes et de les faire avancer d'une colonne à l'autre.
✅ La méthode encourage une **amélioration continue** en adaptant les règles aux besoins de l'équipe.

Kanban est une méthode simple mais puissante, idéale pour mieux organiser ton travail et celui de ton équipe ! 🚀

Le Daily Stand-Up

> *Dans un projet Agile, la communication régulière est essentielle pour avancer efficacement. Le **Daily Stand-Up** est une réunion courte et structurée qui permet à l'équipe de **se synchroniser chaque jour**, d'identifier les blocages et de garder un rythme de travail constant.*

Qu'est-ce qu'un Daily Stand-Up ?

Le Daily Stand-Up est une **réunion quotidienne très courte** (en général 10 à 15 minutes), tenue **debout** pour éviter qu'elle ne s'éternise.

Son objectif n'est pas de résoudre les problèmes en détail, mais de :

- partager l'avancement,
- signaler les difficultés,
- ajuster l'organisation de la journée.

C'est un outil de **pilotage quotidien**, pas une réunion de reporting formel.

Les trois questions clés

Chaque membre de l'équipe répond brièvement à trois questions simples :

- **Qu'ai-je fait depuis la dernière réunion ?**
- **Que vais-je faire aujourd'hui ?**
- **Ai-je un blocage ou une difficulté ?**

Ces questions permettent de rendre le travail visible et d'anticiper les problèmes avant qu'ils ne deviennent critiques.

Règles importantes

Pour que le Daily Stand-Up soit efficace, certaines règles doivent être respectées :

- durée courte (maximum 15 minutes),
- pas de débat technique détaillé,
- pas de justification longue,
- tout le monde participe,
- les problèmes complexes sont traités **après**, avec les personnes concernées.

Le respect de ces règles garantit que la réunion reste utile et non chronophage.

À quoi sert vraiment le Daily Stand-Up ?

Le Daily Stand-Up permet de :

- aligner l'équipe sur les priorités du jour,
- détecter rapidement les blocages,
- favoriser l'entraide,
- éviter les mauvaises surprises en fin de sprint ou de projet,
- maintenir un rythme de travail régulier.

Ce n'est pas un contrôle, mais un **outil de coordination**.

Exemple concret

Projet web en équipe :

- "J'ai terminé la page de connexion."
- "Aujourd'hui, je commence la gestion des rôles."
- "Je suis bloqué sur une erreur de base de données."

Grâce à cette information, l'équipe peut immédiatement :

- proposer de l'aide,
- ajuster la répartition du travail,
- éviter un retard.

Erreurs fréquentes

- transformer le Daily en réunion de résolution de problèmes,
- parler uniquement au chef de projet au lieu de parler à l'équipe,
- dépasser largement le temps prévu,
- ne pas signaler les blocages par peur ou par habitude.

Un Daily Stand-Up inefficace fait perdre du temps au lieu d'en faire gagner.

Synthèse

Ce qu'il faut retenir :

- le Daily Stand-Up est une réunion **courte et quotidienne**,
- il sert à **se synchroniser**, pas à tout expliquer,
- les trois questions structurent l'échange,
- les blocages doivent être signalés immédiatement,
- c'est un outil clé pour garder un projet **visible, fluide et maîtrisé**.

Bien utilisé, le Daily Stand-Up améliore la communication, réduit les retards et renforce le travail d'équipe.

Questionnaire

> *Vérifions ta compréhension du daily standup... aussi appelé Daily Scrum.*

Consigne Réponds aux questions suivantes en t'appuyant sur la vidéo visionnée. Les réponses doivent être courtes et précises.

Questions

1. **Quel est l'objectif principal du Daily Stand-Up ?** ☐ Faire un rapport au chef de projet ☐ Synchroniser l'équipe sur l'avancement du travail ☐ Résoudre tous les problèmes techniques ☐ Planifier tout le sprint
2. **Pourquoi le Daily Stand-Up est-il volontairement très court (10–15 minutes) ?** Explique l'intérêt de cette contrainte de temps.
3. **Quelles sont les trois questions auxquelles chaque membre de l'équipe doit répondre pendant le Daily ?** (Réponds en reformulant, pas mot pour mot.)
4. **Pourquoi dit-on que le Daily Stand-Up est une réunion "pour l'équipe" et non un reporting pour un responsable ?** Explique en une ou deux phrases.
5. **Que faut-il faire si un problème complexe ou technique est identifié pendant le Daily ?** ☐ Le résoudre immédiatement pendant la réunion ☐ L'ignorer ☐ En parler après la réunion avec les personnes concernées ☐ Le noter pour la fin du projet
6. **Cite un bénéfice concret du Daily Stand-Up pour le bon déroulement d'un projet.** (Exemple attendu : gestion des blocages, coordination, anticipation…)

Make/Buy/Reuse (MBR)

> *Dans un projet, tu dois souvent décider **comment** réaliser une fonctionnalité : la fabriquer toi-même, l'acheter, ou la réutiliser. MBR est un modèle simple qui t'aide à faire ce choix de manière logique et défendable.*

Pourquoi MBR est utile

MBR t'évite de prendre une décision "au feeling". Tu compares des options avec des critères concrets (temps, coût, risques…), puis tu choisis la solution la plus adaptée au projet.

Les trois options

Make (faire)

Tu développes la solution toi-même.

Tu choisis **Make** quand :

- tu as un besoin très spécifique (valeur ajoutée du projet),
- tu veux un contrôle maximal (fonctionnement, données, sécurité),
- tu as les compétences et le temps.

Attention : **Make** augmente souvent le risque de retard et de bugs, et demande plus de maintenance.

Buy (acheter)

Tu utilises un produit ou un service existant (licence, SaaS, API payante).

Tu choisis **Buy** quand :

- c'est un besoin standard (authentification, paiement, emailing…),
- tu veux gagner du temps,
- tu acceptes les contraintes d'un fournisseur (prix, limites, contrat, dépendance).

Reuse (réutiliser)

Tu réutilises un composant existant : code interne déjà validé, bibliothèque open-source, template, module du framework.

Tu choisis **Reuse** quand :

- c’est compatible avec ton besoin,
- c’est déjà testé et documenté,
- la licence et la sécurité sont acceptables.

Comment décider rapidement

Pour comparer Make / Buy / Reuse, utilise cette grille (simple mais efficace) :

- **Délai** : est-ce que ça te permet de livrer à temps ?
- **Coût total** : développement + maintenance + abonnements (pas seulement “gratuit/payant”).
- **Risque technique** : est-ce facile à intégrer et à faire fonctionner ?
- **Maintenance** : qui corrige et met à jour dans 6 mois ?
- **Conformité** : licence, RGPD, données hébergées où et par qui ?

Règle pratique : si c’est **standard**, tu vas souvent vers **Buy** ou **Reuse**. Tu gardes **Make** pour ce qui rend ton projet unique.

Exemple concret (application web)

Besoin : “Les utilisateurs doivent se connecter et récupérer leur mot de passe.”

- **Make** : tu codes tout (hash, tokens, reset, protections…).
- **Buy** : tu relies un service d’identité externe.
- **Reuse** : tu utilises un module d’authentification éprouvé (framework/bibliothèque).

Décision fréquente : **Reuse**, car tu avances vite tout en restant sur une solution robuste.

Synthèse

Ce qu’il faut retenir :

- **Make** = contrôle maximal, mais plus long et plus risqué.
- **Buy** = plus rapide, mais dépendance et contraintes.
- **Reuse** = souvent le meilleur compromis si c’est compatible.
- Tu décides avec des critères simples : **délai, coût total, risque, maintenance, conformité**.
- Une bonne décision MBR est **justifiée** et **écrite** (trace de décision).

Études de cas — Décisions MBR en situation réelle

Les études de cas suivantes montrent comment **Make / Buy / Reuse** est utilisé dans de vrais projets informatiques. Chaque cas met en évidence le **contexte**, le **choix effectué** et la **justification**, comme on l’attend dans une gestion de projet réaliste.

▸ Cas réel — Application de billetterie pour un événement local

CONTEXTE

Une petite équipe développe une application web pour gérer des événements : création d'événements, vente de tickets et contrôle à l'entrée. Le délai est court (quelques semaines) et le budget limité.

BESOIN ANALYSÉ

Paiement en ligne des billets.

CHOIX MBR

Buy

JUSTIFICATION

- Le paiement est un besoin **standard** et critique.
- Les exigences de sécurité (cartes bancaires, données sensibles) sont élevées.
- Développer un système fiable prendrait trop de temps et serait risqué.

▸ Décision

Utilisation d'un service de paiement externe via API. L'équipe se concentre sur la logique métier (événements, billets, scan).

À RETENIR

Quand un besoin est sensible, normé et complexe, **Buy** réduit fortement les risques.

▸ Cas réel — Site web d'une école avec espace élèves

CONTEXTE

Une école souhaite un site web avec un espace sécurisé pour les élèves : consultation de documents, messages internes, accès par identifiant.

BESOIN ANALYSÉ

Authentification et gestion des comptes utilisateurs.

CHOIX MBR

Reuse

JUSTIFICATION

- Le framework utilisé propose déjà un module d'authentification éprouvé.
- La solution est documentée et maintenue.
- La licence est compatible avec le projet.

DÉCISION

Réutilisation du module existant, avec une configuration adaptée aux besoins de l'école.

À RETENIR

Quand une solution fiable existe déjà et correspond au besoin, **Reuse** est souvent le meilleur compromis.

▸ Cas réel — Logiciel interne de gestion de matériel informatique

CONTEXTE

Un service informatique développe un outil interne pour suivre le matériel (PC, écrans, imprimantes) et les prêts aux utilisateurs.

BESOIN ANALYSÉ

Gestion spécifique des règles internes (types de matériel, procédures, rapports personnalisés).

CHOIX MBR

Make

JUSTIFICATION

- Les outils existants ne correspondent pas aux règles internes.
- Le logiciel n'est pas destiné à être vendu ou généralisé.
- L'équipe possède les compétences nécessaires.

DÉCISION

Développement sur mesure, parfaitement aligné avec les procédures internes.

À RETENIR

Quand le besoin est très spécifique et stratégique, **Make** permet un contrôle total.

▸ Synthèse transversale

Ces trois cas montrent que :

- **Buy** est pertinent pour les fonctionnalités standards et critiques.
- **Reuse** est idéal quand une solution existante est compatible et fiable.
- **Make** est réservé aux besoins spécifiques qui font la valeur du projet.

En gestion de projet, il n'y a pas de "bon choix universel". Une bonne décision MBR est toujours **contextuelle**, **argumentée** et **assumée**.

La matrice Urgent / Important

> *Quand tu travailles sur un projet, tout semble souvent “urgent”. La matrice **Urgent / Important** est un outil simple qui t’aide à **prioriser intelligemment** tes tâches et à éviter de perdre du temps sur ce qui compte peu.*

Cette matrice est aussi appelée **matrice d’Eisenhower**, ancien président des États-Unis, qui résumait ainsi la gestion du temps :

> *Ce qui est important est rarement urgent, et ce qui est urgent est rarement important.*

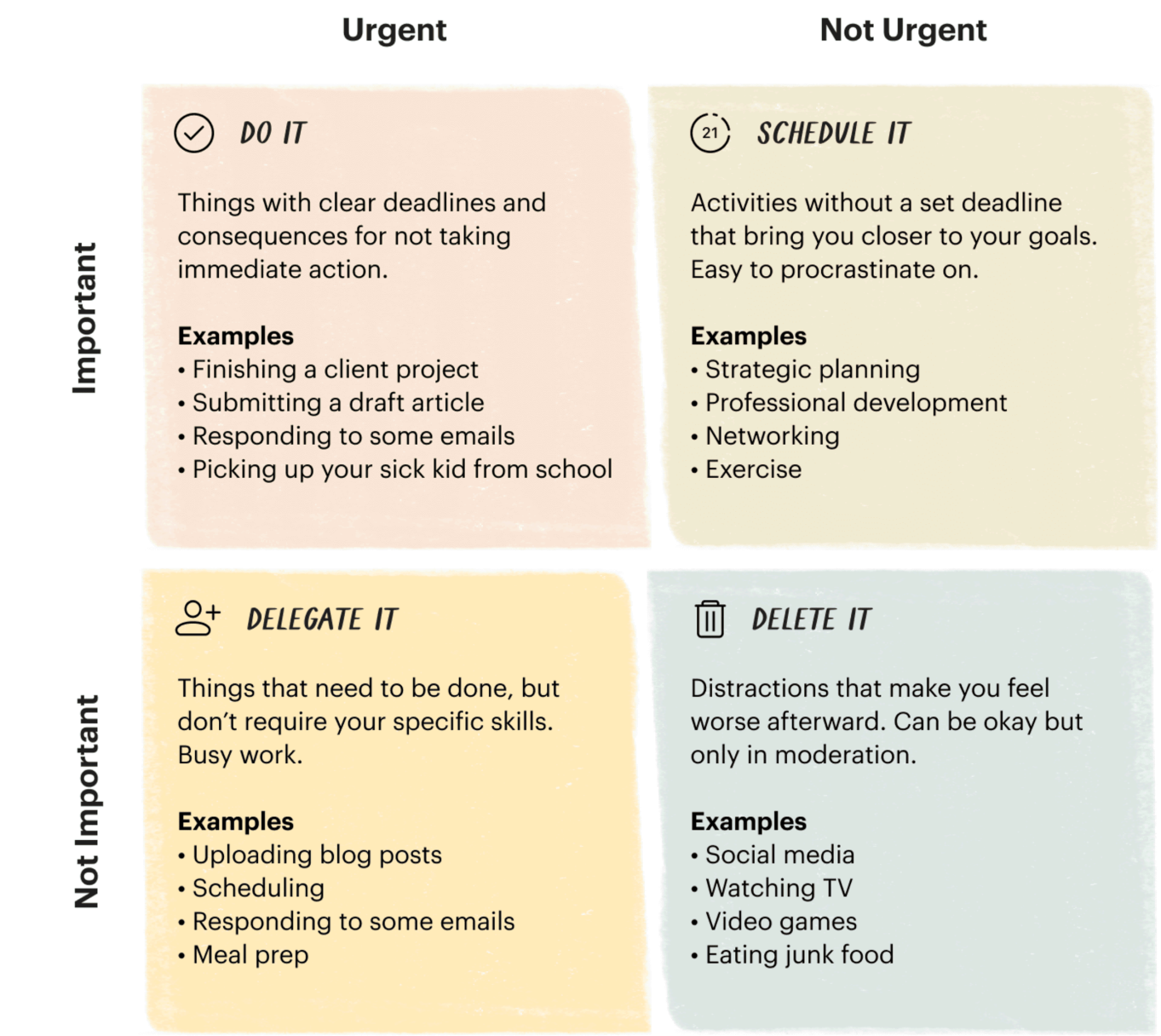
The Urgent Important Matrix Quadrants



Four Quadrants of Productivity

	Urgent	Not Urgent
Important	**Q1** — 5% of Your Time **Fire Fighting** Crisis Production Down Lost Business	**Q2** — 75% of Your Time **Productive Time** Strategy / Vision Planning / Prevention New Opportunities
Not Important	**Q3** — 15% of Your Time **Distraction** Other People's Priorities Reactionary Requests Walk-Ins / Ad-Hoc	**Q4** — 5% of Your Time **Down Time** Busy Work Socializing Entertainment

Eisenhower Quadrants of Productivity



Le principe de base

Chaque tâche peut être évaluée selon **deux critères** :

- **Urgent** : doit être fait tout de suite, il y a une échéance proche ou une conséquence immédiate.
- **Important** : contribue réellement aux objectifs du projet (qualité, réussite, apprentissage).

En croisant ces deux critères, on obtient **4 catégories**.

Les quatre quadrants

Important et urgent — À faire immédiatement

Ce sont les tâches critiques.

Exemples :

- bug bloquant avant une remise,
- serveur en panne,
- rendu de projet aujourd'hui.

Ce sont des tâches **prioritaires absolues**. Trop de tâches dans ce quadrant indique souvent un manque d'anticipation.

Important mais pas urgent — À planifier

Ce sont les tâches essentielles pour la réussite du projet, mais sans pression immédiate.

Exemples :

- conception,
- documentation,
- tests,
- apprentissage d'une nouvelle technologie.

C'est le **quadrant le plus stratégique**. Un bon gestionnaire de projet cherche à passer le plus de temps possible ici.

Urgent mais pas important — À déléguer ou limiter

Ces tâches demandent une réaction rapide mais apportent peu de valeur.

Exemples :

- interruptions fréquentes,
- demandes mineures,
- messages non essentiels.

Si possible, on les **réduit**, on les **automatise** ou on les **délègue**.

Ni urgent ni important — À éliminer

Ces tâches n'apportent rien au projet.

Exemples :

- distractions,
- perfectionnisme inutile,

- tâches “par habitude”.

Elles doivent être **supprimées ou fortement limitées**.

Comment utiliser la matrice en pratique

1. Liste toutes tes tâches.
2. Pour chaque tâche, pose deux questions :
    - Est-ce **urgent** ?
    - Est-ce **important** ?
3. Place la tâche dans le bon quadrant.
4. Agis selon la règle :
    - faire,
    - planifier,
    - limiter,
    - supprimer.

Astuce projet : fais cet exercice **au début d’une séance de travail** ou **au début d’une semaine**.

Exemple concret (projet informatique)

Projet web à rendre dans 2 semaines :

- Corriger un bug qui empêche l’app de démarrer → **important & urgent**
- Concevoir la structure de la base de données → **important & non urgent**
- Répondre à un message non prioritaire → **urgent & non important**
- Modifier des couleurs sans impact fonctionnel → **ni urgent ni important**

La matrice t’aide à **justifier tes choix** et à mieux organiser ton temps.

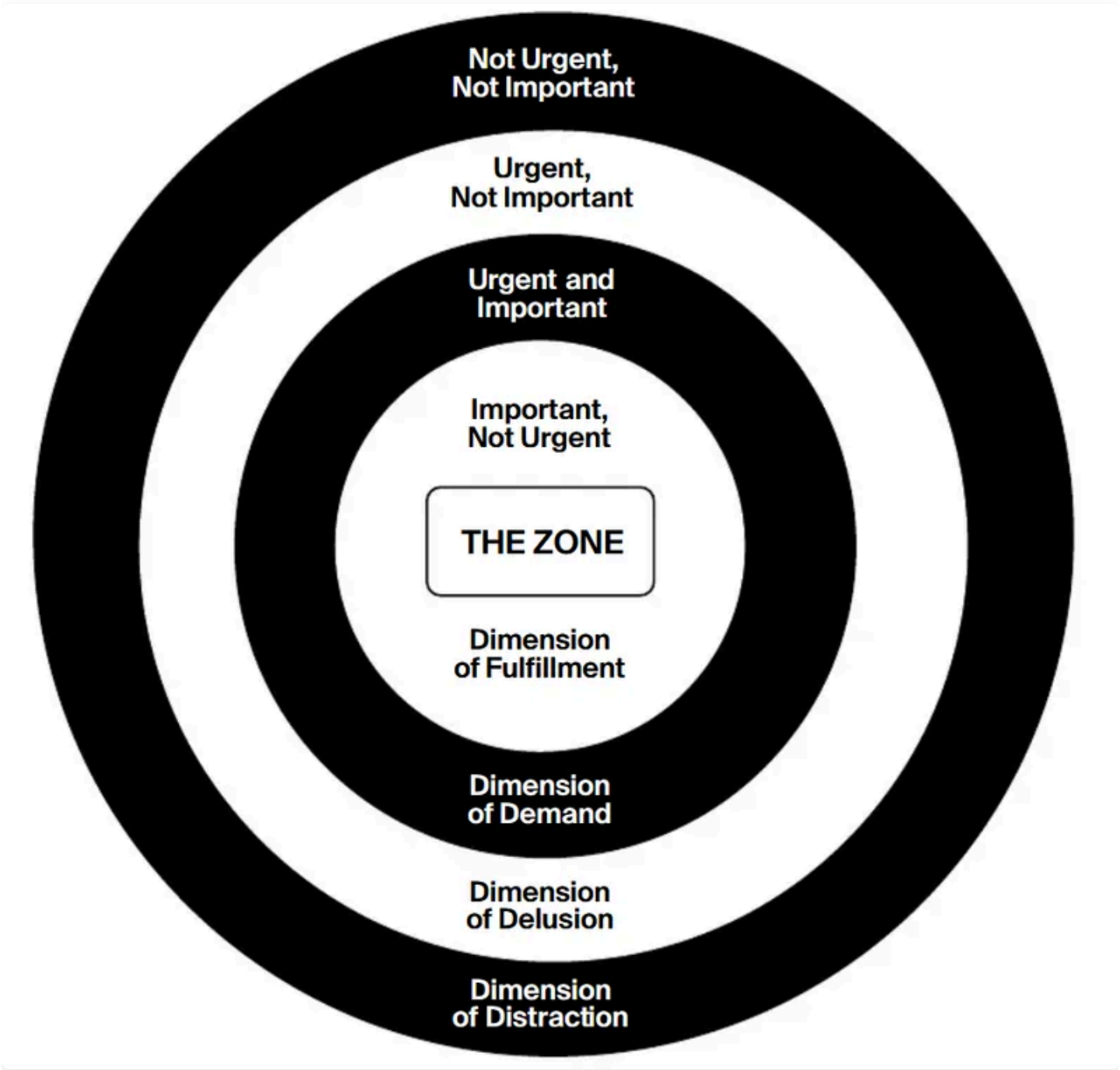
Synthèse

Ce qu’il faut retenir :

- Urgent ≠ Important.
- Toutes les tâches urgentes ne sont pas importantes.
- Les tâches importantes mais non urgentes doivent être **planifiées**.
- La matrice Urgent / Important est un outil simple, mais très efficace, en gestion de projet comme dans le travail scolaire.

Utilisée régulièrement, elle améliore la **priorisation**, réduit le stress et augmente la qualité du travail rendu.

La version "coaching"



Les Time-Targets

> *En gestion de projet, la qualité des résultats dépend directement de **l'endroit où l'équipe concentre son temps**. La notion de *Time Targets* permet de prendre du recul sur la planification et d'éviter un piège fréquent : passer son temps à gérer l'urgence plutôt qu'à construire ce qui est réellement important.*

En gestion de projet, le problème n'est pas le manque de temps, mais **la mauvaise répartition du temps**.

Les *Time Targets* montrent que les projets les plus efficaces sont ceux qui consacrent volontairement du temps aux tâches **importantes mais non urgentes** : conception, anticipation, qualité.

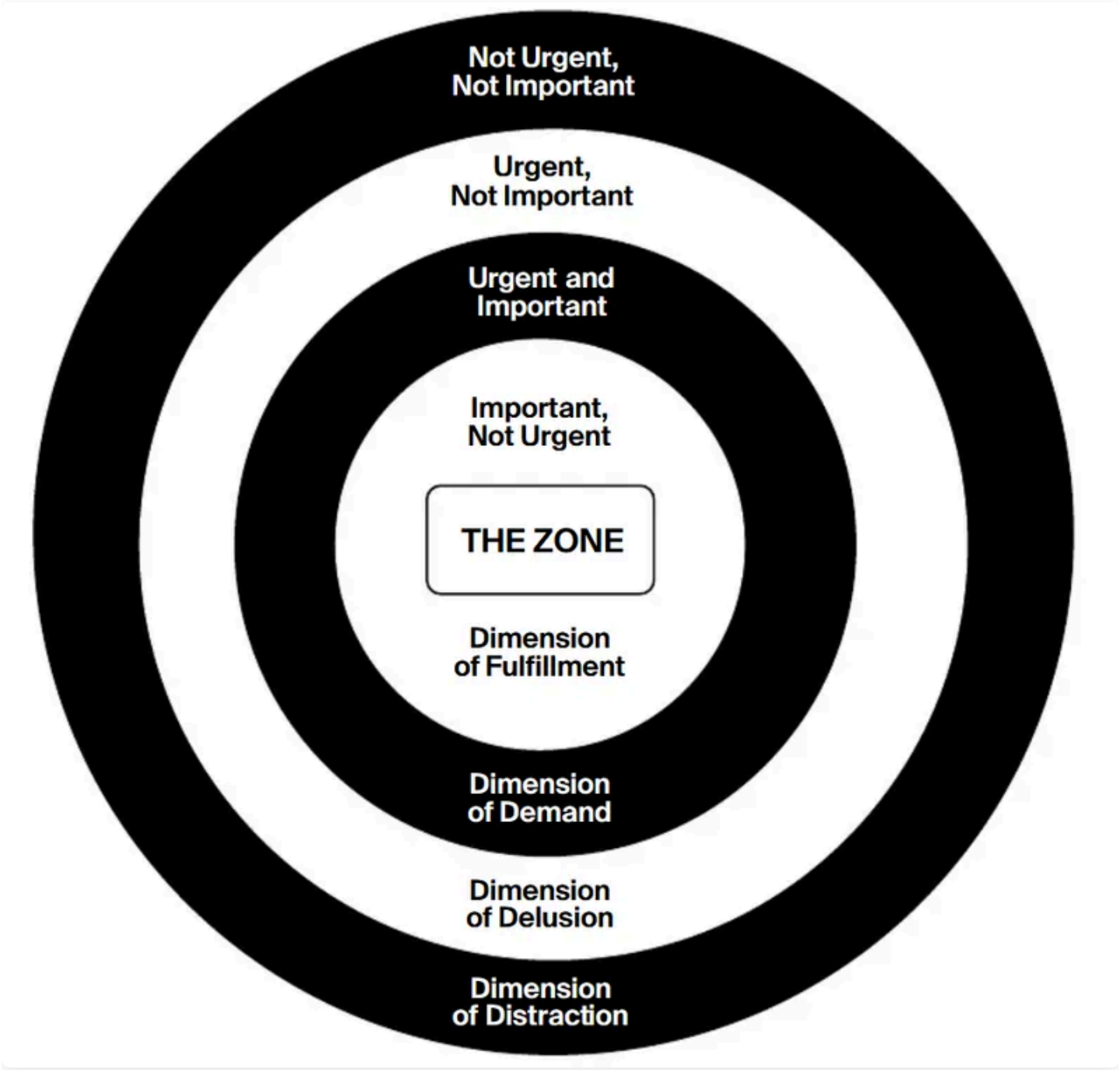
À l'inverse, passer trop de temps à gérer l'urgence, les interruptions ou les distractions donne l'illusion d'avancer, sans réellement faire progresser le projet.

L'objectif d'un bon chef de projet est donc clair : **organiser le travail pour passer le plus de temps possible dans la "Zone"**, là où se construisent les résultats durables.

La question centrale est donc :

> ***Où se situe le focus du projet ?***

Les quatre dimensions du temps



La méthode distingue quatre zones dans lesquelles un projet peut consommer du temps et de l'énergie.

The Zone — Important, mais pas urgent

Dimension de l'accomplissement

C'est la zone idéale en gestion de projet. Elle regroupe toutes les activités **très importantes**, mais qui **n'exigent pas une réaction immédiate** :

- analyse des besoins,
- conception,

- anticipation des risques
- amélioration de la qualité
- documentation
- tests préventifs.

Ces actions renforcent le projet sur le long terme, mais sont souvent repoussées par manque de temps. Pourtant, travailler régulièrement dans cette zone permet de **réduire fortement les urgences futures**.

En projet, l'urgence n'est pas une fatalité : c'est une contrainte artificielle qui doit être maîtrisée.

> *Si on ne fait pas attention à ce sur quoi on se focalise, la "Zone" contiendra souvent toutes ces actions qu'on arrive pas à commencer parce qu'on est "tellement occupé"*

Dimension of Demand — Urgent et important

Gestion des exigences immédiates

Cette zone contient les tâches à la fois urgentes et importantes :

- incidents bloquants,
- livrables à rendre immédiatement,
- problèmes imprévus qui empêchent l'avancement.

Ces situations doivent être traitées sans délai. Cependant, un projet bien piloté cherche à **limiter le temps passé dans cette zone** en anticipant ce qui peut l'être.

Cette zone a aussi souvent tendance à se remplir de tâches importantes qui sont devenues urgentes par manque de temps ou d'anticipation.

Plus un projet est préparé en amont, moins il subit cette dimension.

Dimension of Delusion — Urgent, mais pas important

Illusion de productivité

Cette zone est particulièrement dangereuse.

Elle regroupe des tâches qui donnent l'impression d'avancer parce qu'elles sont urgentes, mais qui **n'apportent que peu de valeur réelle au projet** :

- interruptions constantes,
- demandes secondaires,
- micro-tâches sans impact sur les objectifs,
- réactivité excessive aux sollicitations.

On y travaille beaucoup, mais sans faire progresser le projet de manière significative. C'est une forme d'auto-illusion : être occupé n'est pas synonyme d'être efficace.

Dimension of Distraction — Ni urgent, ni important

Distraction pure

Cette zone correspond à ce qui n'apporte aucune valeur au projet :

- procrastination,
- distractions numériques,
- perfectionnement inutile,
- activités de remplissage.

Y passer trop de temps mène à une stagnation du projet. Le travail avance peu, sans pour autant créer une vraie satisfaction ou un résultat concret.

Lecture projet des *Time Targets*

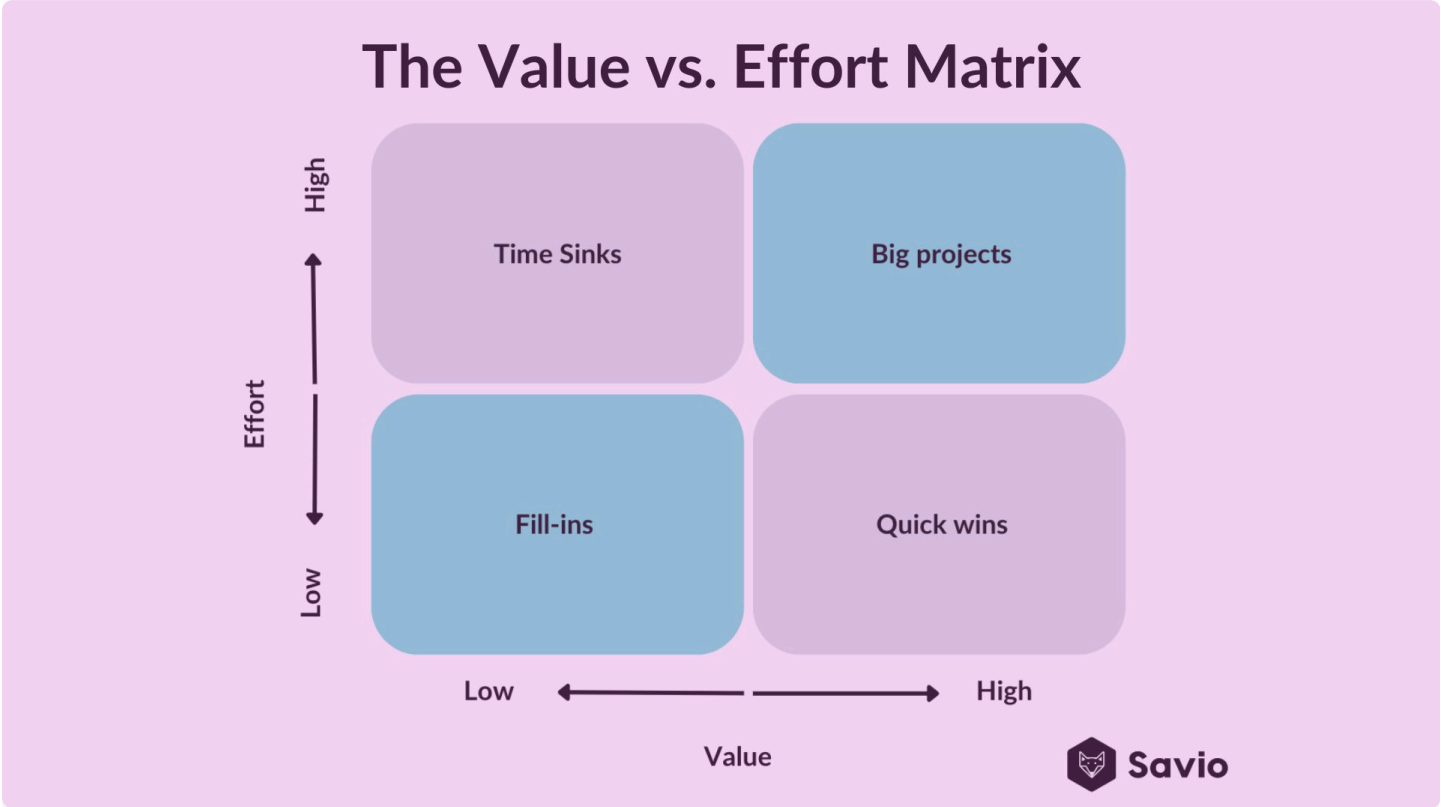
Appliquée à la gestion de projet, cette approche invite à :

- **placer volontairement du temps dans "The Zone"**,
- anticiper les tâches importantes avant qu'elles ne deviennent urgentes,
- réduire les fausses urgences,
- limiter les distractions.

Le véritable pilotage d'un projet consiste à **choisir consciemment où l'on investit son temps**, plutôt que de subir la pression du court terme.

La matrice Valeur / Effort

> *Dans un projet, tu ne peux pas tout faire en même temps. La matrice **Valeur / Effort** est un outil simple et très utilisé en gestion de projet Agile pour **décider quoi faire en priorité**. Elle t'aide à concentrer ton temps sur ce qui apporte le plus de valeur, sans gaspiller d'énergie sur des tâches peu rentables.*



Le principe

Chaque tâche, fonctionnalité ou idée est évaluée selon **deux critères** :

- **Valeur** : à quel point cela est utile pour l'utilisateur, le client ou le projet.
- **Effort** : le temps, la complexité et les difficultés nécessaires pour la réaliser.

En croisant ces deux critères, tu peux comparer les tâches **entre elles**, au lieu de travailler "au hasard" ou sous la pression.

Les quatre zones de la matrice

Forte valeur – Faible effort

C'est la priorité absolue.

Ces tâches apportent beaucoup de valeur pour peu de travail :

- fonctionnalité simple mais très utile,
- correction rapide d'un problème important,
- amélioration visible pour l'utilisateur.

👉 **À faire en premier.** Ce sont les meilleurs investissements de temps.

Forte valeur – Fort effort

À planifier.

Ces tâches sont importantes pour le projet, mais demandent beaucoup de travail :

- grosse fonctionnalité centrale,
- refonte technique,
- étape clé du projet.

👉 **À découper, planifier et anticiper.** Elles ne doivent pas être improvisées.

Faible valeur – Faible effort

Optionnel.

Ces tâches sont faciles à faire, mais apportent peu :

- petits ajouts de confort,
- détails esthétiques secondaires.

👉 **À faire seulement s'il reste du temps.**

Faible valeur – Fort effort

À éviter.

Ces tâches demandent beaucoup d'énergie pour peu de bénéfices :

- fonctionnalités inutiles,
- complexité non justifiée,
- perfectionnisme excessif.

👉 **À éliminer du projet.**

Comment l'utiliser en pratique

1. Liste les tâches ou fonctionnalités du projet.
2. Évalue chacune selon :
    - Valeur (faible → élevée),

    - Effort (faible → élevé).
3. Place-les dans la matrice.
4. Décide collectivement des priorités.

Important : il n'y a pas de mesure "parfaite". Ce qui compte, c'est la **discussion et la justification des choix**.

Exemple concret

Projet web :

- Connexion utilisateur simple → **forte valeur / faible effort**
- Animation complexe d'introduction → **faible valeur / fort effort**
- Mise en place de la base de données → **forte valeur / fort effort**
- Changer une couleur secondaire → **faible valeur / faible effort**

La matrice permet de **voir immédiatement** ce qui doit être fait en priorité.

Synthèse

Ce que tu dois retenir :

- La matrice Valeur / Effort aide à **prioriser intelligemment**.
- On privilégie ce qui apporte **beaucoup de valeur pour peu d'effort**.
- Toutes les tâches ne se valent pas.
- En gestion de projet Agile, **faire moins mais mieux** est souvent la clé de la réussite.

C'est un outil simple, mais extrêmement efficace pour prendre de bonnes décisions dans un projet réel.

La méthode MoSCoW

> *Dans un projet, toutes les fonctionnalités n'ont pas la même importance. La méthode **MoSCoW** est une technique de priorisation simple et très utilisée (notamment en Agile) pour **clarifier ce qui est indispensable, ce qui est souhaitable et ce qui ne sera pas fait**. Elle permet de cadrer le périmètre du projet, d'éviter les malentendus et de mieux gérer le temps et le budget.*



Le principe

MoSCoW classe les fonctionnalités (ou exigences) en **quatre catégories**, selon leur **importance réelle pour la réussite du projet**.

L'objectif n'est pas de tout faire, mais de **faire en priorité ce qui est essentiel**.

Les quatre catégories MoSCoW

Must have — Indispensable

Ce sont les fonctionnalités **obligatoires**.

Sans elles :

- le projet ne fonctionne pas,
- ou ne répond pas au cahier des charges.

Exemples :

- sauvegarde des données,
- accès sécurisé,
- fonctionnalité principale du projet.

👉 **Un projet sans les “Must” est un projet raté.**

Should have — Important

Ces fonctionnalités apportent une **valeur forte**, mais le projet peut fonctionner sans elles à court terme.

Exemples :

- amélioration de l’ergonomie,
- fonctionnalités avancées,
- options utiles mais non critiques.

👉 À faire **si possible**, après les Must.

Could have — Optionnel

Ces fonctionnalités sont agréables, mais **non nécessaires**.

Exemples :

- options de confort,
- éléments esthétiques secondaires,
- petites améliorations non essentielles.

👉 À réaliser **s’il reste du temps et du budget**.

Would / Won’t have — Non prévu

Dans la méthode originale, le **W** signifie *Would have*. En pratique, en gestion de projet (et très souvent en contexte scolaire ou agile), on utilise plutôt **Won’t have**.

Cela correspond à :

- des idées intéressantes,
- mais que l'on **n'a pas le temps ou le budget de réaliser**,
- ou qui sont volontairement exclues du périmètre.

👉 **Won't = pas pour cette version du projet.**

C'est une décision consciente, pas un oubli.

Pourquoi utiliser MoSCoW

La méthode MoSCoW permet de :

- clarifier les priorités dès le départ,
- éviter le "tout est important",
- sécuriser les délais,
- mieux gérer les attentes du client ou de l'enseignant,
- garder une vision réaliste du projet.

Elle est particulièrement efficace en début de projet ou lors de la planification d'une itération.



Exemple concret

Projet d'application web :

- Authentification utilisateur → **Must**
- Gestion des rôles (admin/utilisateur) → **Should**
- Thème sombre personnalisable → **Could**
- Intelligence artificielle de recommandation → **Won't**

La méthode rend les choix **lisibles et assumés**.

Bonnes pratiques

- Il doit toujours y avoir des éléments dans **Won't**.
- Trop de **Must** = projet irréaliste.
- Les catégories peuvent évoluer au fil du projet.
- La priorisation se fait **collectivement et de manière argumentée**.

Synthèse

Ce qu'il faut retenir :

- MoSCoW est une méthode simple de priorisation.
- **Must** = indispensable.
- **Should** = important.
- **Could** = optionnel.
- **Won't** = volontairement exclu (manque de temps ou de budget).
- MoSCoW aide à livrer un projet **réaliste et maîtrisé**.

C'est un outil essentiel pour apprendre à **faire des choix** en gestion de projet, plutôt que de subir les contraintes.

Les types de bugs et leur gravité

> *Dans un projet informatique, tous les bugs ne se valent pas. Certains bloquent complètement le travail, d'autres sont gênants, et certains n'ont qu'un impact mineur. Classer les bugs par **niveau de gravité** est essentiel pour **prioriser les corrections**, organiser le travail de l'équipe et respecter les délais.*

Pourquoi classer les bugs

Sans classification, tout bug semble urgent. En réalité, corriger un bug graphique mineur avant un bug bloquant est une **mauvaise décision de gestion de projet**.

La classification permet de :

- décider **quoi corriger en premier**,
- communiquer clairement dans l'équipe,
- éviter les débats subjectifs ("ce bug est grave / pas grave").

Bug bloquant (Blocking / Critical)

Un bug bloquant empêche totalement l'utilisation du logiciel ou l'avancement du projet.

Caractéristiques :

- l'application ne démarre pas,
- une fonctionnalité essentielle ne fonctionne pas du tout,
- perte ou corruption de données,
- impossibilité de livrer ou de tester.

Exemples :

- crash au lancement,
- impossible de se connecter,
- paiement impossible alors que c'est la fonctionnalité centrale.

👉 **Priorité absolue.** Un bug bloquant doit être corrigé immédiatement.

Bug majeur (Major)

Un bug majeur affecte fortement le fonctionnement, mais **une solution de contournement existe**.

Caractéristiques :

- fonctionnalité importante partiellement cassée,
- comportement incorrect mais non bloquant,
- impact fort sur l'expérience utilisateur.

Exemples :

- un formulaire fonctionne, mais avec des erreurs fréquentes,
- une fonctionnalité clé est lente ou instable,
- affichage incorrect qui gêne l'utilisation.

👉 **À corriger rapidement**, juste après les bugs bloquants.

Bug mineur (Minor)

Un bug mineur a un impact limité sur l'utilisation du logiciel.

Caractéristiques :

- le logiciel reste utilisable,
- l'erreur est surtout esthétique ou secondaire,
- aucun impact sur les données ou la logique principale.

Exemples :

- texte mal aligné,
- faute d'orthographe,
- couleur incorrecte sur un bouton.

👉 **À corriger si le temps le permet.**

Bug trivial (Cosmetic / Trivial)

Ce type de bug n'a quasiment **aucun impact fonctionnel**.

Caractéristiques :

- purement visuel,
- très faible gêne pour l'utilisateur,
- n'affecte ni la compréhension ni l'usage.

Exemples :

- icône légèrement décalée,
- animation imparfaite,
- détail graphique mineur.

👉 **Souvent ignoré ou corrigé en fin de projet.**

Autres classifications courantes

Selon les équipes, on peut aussi rencontrer :

- **Bug de performance** : lenteur, consommation excessive de ressources.
- **Bug de sécurité** : faille, accès non autorisé, fuite de données.

- **Bug de compatibilité** : problème selon le navigateur, l'OS ou le matériel.
- **Bug de logique** : le programme fonctionne, mais pas comme attendu.

Ces catégories décrivent la **nature du bug**, tandis que *bloquant / majeur / mineur* décrit sa **gravité**.

Gravité vs priorité

Attention :

- **Gravité** = impact réel du bug.
- **Priorité** = ordre dans lequel on décide de le corriger.

Un bug mineur peut devenir prioritaire s'il touche beaucoup d'utilisateurs ou s'il est très visible.

Synthèse

Ce qu'il faut retenir :

- Tous les bugs n'ont pas la même gravité.
- **Bloquant** : empêche de travailler ou d'utiliser le logiciel.
- **Majeur** : gêne fortement, mais sans blocage total.
- **Mineur** : impact limité.
- **Trivial** : impact négligeable.
- Classer les bugs permet de **corriger efficacement** et de **tenir les délais**.

Bien gérer les bugs, ce n'est pas tout corriger tout de suite, c'est **corriger ce qui compte vraiment en premier**.

Projet d'examen — Système multi-langages (C · Python · Flask · C#)

> *Projet d'examen intégrateur en 5 composantes : une DLL en C, un module Python testé avec Pytest, une API REST Flask avec dashboard Chart.js, et une interface C# — à réaliser individuellement sur l'un des 6 sujets au choix.*

Tu vas réaliser un projet complet qui fait collaborer **quatre langages** autour d'un même domaine métier : une bibliothèque C compilée en DLL, un module Python qui l'exploite, une API Flask qui sert les données, et une interface C# qui les affiche — le tout complété d'un dashboard web et d'un générateur de données.

🎯 Objectifs

À la fin de ce projet, tu seras capable de :

1. Compiler un fichier C en bibliothèque partagée (`.dll`) et l'appeler depuis Python via `ctypes` .
2. Écrire un module Python testable avec **Pytest** (minimum 10 tests PASSED).
3. Concevoir une API REST avec **Flask** exposant des données au format JSON.
4. Créer un **dashboard web** interactif avec Chart.js consommant l'API.
5. Développer une interface **C# Windows Forms** communicant avec l'API via `HttpClient` .
6. Mesurer et comparer les performances de trois approches de calcul (Python pur, ctypes scalaire, ctypes batch).

Partie 1 — Architecture générale

La chaîne technique relie 5 composantes. Chaque couche ne communique qu'avec celle directement en dessous ou au-dessus : **le C n'est jamais appelé directement par Flask**.

Tu trouveras le **schéma d'architecture générale** dans l'encart "Pour aller plus loin" à droite.

```
calculs.c  →  gcc -shared -o calculs.dll calculs.c -lm
                        ↓  ctypes
              module.py  ←──  pytest tests_module.py
                        ↓
              app.py  (Flask)
                ├── /api/...       ← JSON ←── GUI C# (HttpClient)
                └── /dashboard/   ← HTML + Chart.js (lecture seule)

              generator.py  ──POST──►  /api/...
```

> 💡 ***Règle d'or :*** *tout calcul reste dans la DLL ou le module Python. La GUI C# et le dashboard affichent — ils ne calculent rien.*

Compilation de la DLL

```
# Linux / macOS
gcc -shared -fPIC -o calculs.so calculs.c -lm

# Windows (MinGW)
gcc -shared -o calculs.dll calculs.c -lm

# Vérifier les fonctions exportées (Linux)
nm -D calculs.so | grep " T "
```

Chargement depuis Python

```
import ctypes

lib = ctypes.CDLL("./calculs.dll")

# Déclarer les types — OBLIGATOIRE pour éviter les erreurs silencieuses
lib.calc_bmi.restype  = ctypes.c_float
lib.calc_bmi.argtypes = [ctypes.c_float, ctypes.c_float]

# Appel scalaire
bmi = lib.calc_bmi(ctypes.c_float(70.0), ctypes.c_float(1.75))

# Appel batch (tableau de N valeurs)
N = 1_000_000
weights = (ctypes.c_float * N)(*liste_poids)
heights = (ctypes.c_float * N)(*liste_tailles)
results = (ctypes.c_float * N)()
lib.calc_bmi_batch(weights, heights, results, N)
```

Partie 2 — Les 6 projets

Choisis **un seul** projet. Télécharge le cahier des charges complet via les liens ci-dessus.

#	PROJET	DOMAINE	FONCTIONS C (EXEMPLES)
1	Station de mesures	Physique / capteurs	`to_fahrenheit` , `is_anomaly`
2	Suivi fitness	Santé / sport	`calc_bmi` , `calories_burned`
3	Tournoi ELO	Jeux / classement	`expected_score` , `new_elo`
6	Caisse POS	Commerce	`calc_vat` , `render_change`
8	Notes scolaires	Éducation	`weighted_avg` , `std_deviation`
10	Calculatrice scientifique	Mathématiques	`calc_gcd` , `calc_factorial`

💡 ***Pour les 5e :*** *persistance en JSON (dossier* `data/` *).* ***Pour les 6e :*** *base de données SQL (MySQL via Laragon ou SQLite).*

Partie 3 — Les 5 composantes en détail

▸ 3.1 — Module C (`calculs.c` → `calculs.dll`)

Deux niveaux de fonctions pour chaque projet :

TYPE	UTILISATION	EXEMPLE
Scalaire	Application normale	`float calc_bmi(float w, float h)`
Batch	Benchmark uniquement	`void calc_bmi_batch(float* w, float* h, float* out, int n)`

▸ 3.2 — Module Python (`module.py`)

Wrapper `ctypes` + fonctions métier testables. Flask l'importe directement — pas de logique dans `app.py` .

```
# module.py — exemple pour le projet 1
import ctypes, os

_lib = ctypes.CDLL(os.path.join(os.path.dirname(__file__), "calculs.dll"))
_lib.to_fahrenheit.restype  = ctypes.c_float
_lib.to_fahrenheit.argtypes = [ctypes.c_float]

def to_fahrenheit(celsius: float) -> float:
    return _lib.to_fahrenheit(ctypes.c_float(celsius))

def validate_measure(value: float, sensor_type: str) -> bool:
    ranges = {"temperature": (-50, 100), "pressure": (800, 1100), "humidity": (0, 100)}
    lo, hi = ranges.get(sensor_type, (float('-inf'), float('inf')))
    return lo <= value <= hi
```

▸ 3.3 — API Flask (`app.py`)

Toutes les réponses respectent le format uniforme :

```
from flask import Flask, jsonify, request
import module  # le module Python local

app = Flask(__name__)

@app.route("/api/measures", methods=["GET"])
def get_measures():
    data = module.load_all()  # lit JSON ou SQL
    return jsonify({"status": "ok", "data": data})

@app.route("/api/measures", methods=["POST"])
def add_measure():
    body = request.json
    if not module.validate_measure(body["value"], body["type"]):
        return jsonify({"status": "error", "message": "Valeur hors plage"}), 400
    result = module.save_measure(body)
    return jsonify({"status": "ok", "data": result}), 201
```

▸ 3.4 — Dashboard Web (`/dashboard`)

Accessible sur `http://localhost:5000/dashboard`. Lecture seule, 3 graphiques Chart.js, rafraîchissement automatique.

```
<!-- templates/dashboard.html -->
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<script>
  async function loadData() {
    const res  = await fetch('/api/measures/stats');
    const json = await res.json();
    // Mise à jour des graphiques avec json.data
  }
  loadData();
  setInterval(loadData, 30_000); // rafraîchit toutes les 30 secondes
</script>
```

▸ 3.5 — Interface C# (`MainForm.cs`)

Toute la communication passe par `HttpClient`. Aucun calcul dans le code C#.

```
using System.Net.Http;
using System.Text;
using System.Text.Json;

private readonly HttpClient _client = new()
    { BaseAddress = new Uri("http://localhost:5000") };

// GET
private async Task<JsonDocument> GetAsync(string path) {
    var resp = await _client.GetAsync(path);
    var json = await resp.Content.ReadAsStringAsync();
    return JsonDocument.Parse(json);
}

// POST
private async Task PostAsync(string path, object body) {
    var json    = JsonSerializer.Serialize(body);
    var content = new StringContent(json, Encoding.UTF8, "application/json");
    await _client.PostAsync(path, content);
}
```

Partie 4 — Générateur de données automatique

Le générateur simule des utilisateurs réels en envoyant des données à ton API. Il utilise **Python +** `requests`.

Active d'abord CORS dans Flask (nécessaire si le générateur tourne dans un autre processus) :

```
pip install flask-cors
```

```
# app.py — ajouter
from flask_cors import CORS
CORS(app)
```

▸ Script `generator.py`

```
"""
Générateur automatique de données — Projets TTINFO 5e/6e
Usage : python generator.py --projet 1 --n 100 --delay 0.3
"""
import requests, random, time, argparse
from datetime import date, timedelta

BASE = "http://localhost:5000"

# ── Générateurs par projet ──────────────────────────────────────

def gen_projet1(n, delay):
    """Station de mesures physiques"""
    types   = ['temperature', 'pressure', 'humidity']
    ranges  = {'temperature':(-10,45), 'pressure':(950,1060), 'humidity':(10,95)}
    units   = {'temperature':'°C', 'pressure':'hPa', 'humidity':'%'}
    for i in range(n):
        t   = random.choice(types)
        lo, hi = ranges[t]
        val = round(random.uniform(lo, hi * (1.15 if random.random()<.08 else 1)), 2)
        r   = requests.post(f"{BASE}/api/measures", json={"type":t,"value":val,"unit":units
        _log(i+1, n, r.status_code, f"{t}={val}")
        time.sleep(delay)

def gen_projet2(n, delay):
    """Suivi sportif & fitness"""
    types = ['course', 'vélo', 'natation', 'musculation', 'yoga']
    mets  = {'course':8.0, 'vélo':6.0, 'natation':7.0, 'musculation':4.0, 'yoga':2.5}
    for i in range(n):
        t  = random.choice(types)
        dur = random.randint(20, 90)
        r  = requests.post(f"{BASE}/api/sessions", json={
            "type": t, "duration_min": dur,
            "met_value": mets[t] + random.uniform(-0.5, 0.5),
            "date": str(date.today() - timedelta(days=random.randint(0,30)))
        })
        _log(i+1, n, r.status_code, f"{t} {dur}min")
        time.sleep(delay)

def gen_projet3(n, delay):
    """Tournoi ELO — génère des matchs entre joueurs existants"""
    players = requests.get(f"{BASE}/api/players").json().get("data", [])
    if len(players) < 2:
        print("Ajoute d'abord au moins 2 joueurs via la GUI.")
        return
```

```
    for i in range(n):
        a, b = random.sample(players, 2)
        res  = random.choice([1.0, 0.5, 0.0])
        r    = requests.post(f"{BASE}/api/matches",
               json={"player_a_id":a["id"],"player_b_id":b["id"],"result":res})
        _log(i+1, n, r.status_code, f"{a['name']} vs {b['name']} → {res}")
        time.sleep(delay)

def gen_projet6(n, delay):
    """Caisse POS — génère des transactions"""
    products = requests.get(f"{BASE}/api/products").json().get("data", [])
    if not products:
        print("Ajoute d'abord des produits via la GUI.")
        return
    for i in range(n):
        items = [{"product_id": p["id"], "qty": random.randint(1,4)}
                 for p in random.sample(products, k=min(random.randint(1,4), len(products))]
        r = requests.post(f"{BASE}/api/transactions",
            json={"items": items, "amount_given": 50.0})
        _log(i+1, n, r.status_code, f"{len(items)} article(s)")
        time.sleep(delay)

def gen_projet8(n, delay):
    """Notes scolaires"""
    students = requests.get(f"{BASE}/api/students").json().get("data", [])
    courses  = requests.get(f"{BASE}/api/courses").json().get("data", [])
    if not students or not courses:
        print("Ajoute d'abord des élèves et des cours.")
        return
    for i in range(n):
        s = random.choice(students)
        c = random.choice(courses)
        r = requests.post(f"{BASE}/api/grades", json={
            "student_id": s["id"], "course_id": c["id"],
            "value": round(random.gauss(12, 3.5), 1),
            "weight": random.choice([1.0, 1.0, 1.0, 2.0]),
            "date": str(date.today())
        })
        _log(i+1, n, r.status_code, f"{s['name']} — {c['name']}")
        time.sleep(delay)

def gen_projet10(n, delay):
    """Calculatrice scientifique"""
    ops = ["sqrt({})", "power({},{})", "gcd({},{})", "factorial({})", "lcm({},{})"]
    for i in range(n):
        expr = random.choice(ops)
        if expr.count('{}') == 1:
```

```
            expr = expr.format(random.randint(1, 144))
        else:
            a, b = random.randint(1,50), random.randint(1,50)
            expr = expr.format(a, b)
        r = requests.post(f"{BASE}/api/calculate", json={"expression": expr})
        _log(i+1, n, r.status_code, expr)
        time.sleep(delay)

# ── Utilitaire ──────────────────────────────────────────────────────────

def _log(i, n, code, info):
    ok = "✅" if code in (200,201) else "❌"
    print(f"  [{i:>4}/{n}] {ok} HTTP {code} | {info}")

GENS = {1:gen_projet1, 2:gen_projet2, 3:gen_projet3,
        6:gen_projet6, 8:gen_projet8, 10:gen_projet10}

# ── CLI ─────────────────────────────────────────────────────────────────

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Générateur de données TTINFO")
    parser.add_argument("--projet", type=int, choices=[1,2,3,6,8,10], required=True,
                        help="Numéro du projet (1/2/3/6/8/10)")
    parser.add_argument("--n",      type=int,   default=50,
                        help="Nombre d'entrées à générer (défaut: 50)")
    parser.add_argument("--delay",  type=float, default=0.2,
                        help="Délai entre chaque requête en secondes (défaut: 0.2)")
    parser.add_argument("--url",    default="http://localhost:5000",
                        help="URL de base de l'API Flask")
    args = parser.parse_args()

    BASE = args.url
    print(f"\n🚀 Générateur — Projet {args.projet} — {args.n} entrées @ {args.delay}s/req")
    print(f"   API cible : {BASE}\n")
    GENS[args.projet](args.n, args.delay)
    print(f"\n✅ Terminé.")
```

▸ Utilisation

```
# Installer la dépendance
pip install requests

# Générer 100 mesures pour le projet 1, une toutes les 0.3 secondes
python generator.py --projet 1 --n 100 --delay 0.3

# Générer 50 matchs pour le projet 3 (joueurs déjà créés)
python generator.py --projet 3 --n 50 --delay 0.1

# Pointer vers une API distante
python generator.py --projet 6 --n 200 --url http://192.168.1.10:5000
```

> 💡 ***Astuce :*** *lance d'abord ton API Flask (`python app.py`), puis le générateur dans un second terminal. Observe le dashboard se remplir en temps réel.*

Partie 5 — Benchmark de performance

Le fichier `benchmark.py` mesure trois approches sur **10 batches de 1 000 000 calculs**.

SCÉNARIO	DESCRIPTION	RÉSULTAT ATTENDU
A — Python pur	Boucle Python native	Référence (×1.0)
B — ctypes scalaire	1 appel ctypes par itération	**Plus lent** que A (overhead ×N)
C — ctypes batch	1 seul appel, C traite tout	Le plus rapide (×10 à ×20)

```python
# benchmark.py — squelette à compléter
import ctypes, time

lib = ctypes.CDLL("./calculs.dll")
# TODO : déclarer argtypes et restype

N = 1_000_000
NB_BATCHES = 10

def bench(label, fn):
    times = []
    for _ in range(NB_BATCHES):
        t = time.perf_counter()
        fn()
        times.append(time.perf_counter() - t)
    total = sum(times)
    return label, total, total / NB_BATCHES

# TODO : implémenter A(), B(), C()
# TODO : afficher le tableau avec les facteurs ×

# Format de sortie attendu :
# Scénario              Temps total    Moy./batch    Facteur
# Python pur            0.08 s          8.0 ms       ×1.0
# ctypes scalaire       0.83 s         83.0 ms       ×0.1   ⚠️
# ctypes batch          0.006 s         0.6 ms       ×13    ✅
```

Question orale attendue : *"Explique pourquoi le scénario B est plus lent que Python pur."* Réponse attendue : l'overhead `ctypes` (conversion de types, franchissement du GIL) est plus coûteux que le calcul lui-même. La solution : déplacer la **boucle dans le C**, pas uniquement le calcul.

🧪 Consignes de remise

Structure de dossier attendue

```
projet-XXXX/
├── calculs.c              ← code source C
├── calculs.dll            ← DLL compilée
├── module.py              ← wrapper ctypes + fonctions métier
├── tests_module.py        ← 10 tests Pytest (tous PASSED)
├── app.py                 ← API Flask + route /dashboard
├── generator.py           ← générateur de données
├── benchmark.py           ← benchmark 3 scénarios
├── templates/
│   └── dashboard.html     ← dashboard Chart.js
├── data/                  ← (5e seulement) fichiers JSON
│   └── *.json
└── GUI/
    └── ProjetXX.sln       ← solution C# Windows Forms
```

Barème synthétique

COMPOSANTE	POINTS
Module C (scalaire + batch, compile sans erreur)	/4
Module Python (wrapper + fonctions)	/4
Tests Pytest (10/10 PASSED)	/3
API Flask (tous les endpoints)	/4
Dashboard web (3 graphiques Chart.js)	/3
GUI C# Windows Forms	/4
Benchmark (3 scénarios + tableau + explication orale)	/2
Total	**/24**

Exercice 1 — Vérifie ta chaîne technique

Avant de tout développer, vérifie que la chaîne de base fonctionne :

1. Écris `calculs.c` avec **une seule fonction** (ex. `float add(float a, float b)`)
2. Compile-la en DLL
3. Appelle-la depuis Python avec `ctypes` et affiche le résultat
4. Lance Flask et crée un endpoint `GET /test` qui appelle ce module Python
5. Depuis C# : appelle `GET /test` et affiche la réponse dans un Label

Une fois cette mini-chaîne validée, tu peux ajouter les vraies fonctions.

▸ Exercice 2 — Lance le générateur et observe le dashboard

1. Lance ton API Flask
2. Lance le générateur : `python generator.py --projet XX --n 50 --delay 0.5`
3. Ouvre le dashboard dans le navigateur
4. Observe les graphiques se remplir en temps réel
5. Note ce qui se passe si l'API est arrêtée pendant la génération

---

✍️ À retenir

- Le **C** est compilé en DLL et appelé via `ctypes` — jamais directement depuis Flask.
- La **boucle** doit être dans le C (batch) pour de vraies performances — les appels `ctypes` répétés N fois sont plus lents que Python pur.
- Le **module Python** est le seul pont entre la DLL et le reste de l'application.
- Le **dashboard** est lecture seule — il affiche, il ne modifie jamais de données.
- La **GUI C#** ne recalcule rien — elle délègue tout à l'API via `HttpClient`.
- Les **tests Pytest** testent le module Python, ce qui teste indirectement le C.

Suite

Le projet est à remettre complet avec tous les fichiers listés ci-dessus. Prépare une **démonstration de 5 minutes** : lance l'API, le générateur, montre le dashboard en direct, puis la GUI C#. Explique ta chaîne technique et les résultats du benchmark.

LE LANGAGE C-SHARP (C#)

C'est quoi C# ?

C# (prononcé *"C sharp"*) est un langage de programmation créé par **Microsoft** en 2000.

On l'utilise pour créer :

- des applications Windows (logiciels de bureau)
- des jeux vidéo (Unity est entièrement basé sur C#)
- des applications web (ASP.NET)
- des applications mobiles (via .NET MAUI)

C# fait partie de l'écosystème **.NET** — un ensemble d'outils, de bibliothèques et d'environnements d'exécution maintenus par Microsoft.

Compilé vs interprété

C'est une distinction fondamentale en programmation.

Un langage interprété

Le code source est **lu et exécuté ligne par ligne** au moment où le programme tourne, par un programme appelé *interpréteur*.

```
Code source (.py, .php, .js)
        ↓
   Interpréteur        ← lit et exécute en direct
        ↓
    Résultat
```

> *Exemples : Python, PHP, JavaScript*

Un langage compilé

Le code source est **traduit une seule fois** en un fichier exécutable, par un programme appelé *compilateur*. C'est ce fichier qui est ensuite exécuté — plus besoin du code source.

```
Code source (.cs)
        ↓
    Compilateur         ← traduit une fois
        ↓
  Fichier exécutable
        ↓
    Résultat
```

> *Exemples : C, C++, C#, Rust*

Le cas particulier de C# et .NET

C# utilise un mécanisme en deux temps :

```
Code source (.cs)
        ↓
    Compilateur C#
        ↓
   Code IL (Intermediate Language)     ← code "intermédiaire"
        ↓
   CLR – Common Language Runtime       ← exécuté sur la machine
        ↓
    Résultat
```

Le compilateur ne produit pas directement du code machine, mais un **code intermédiaire** (IL) compris par le **CLR**, la machine virtuelle de .NET.

> 💡 *Ce fonctionnement permet à un programme C# de tourner sur Windows, Linux ou macOS **sans recompiler** — à condition que .NET soit installé sur la machine cible.*

Pourquoi c'est important ?

	COMPILÉ (C#)	INTERPRÉTÉ (PYTHON, PHP)
Détection des erreurs	À la **compilation** — avant d'exécuter	À l'**exécution** — en cours de route
Performance	✅ généralement plus rapide	Dépend de l'interpréteur
Fichier livrable	Un `.exe` ou `.dll` autonome	Le code source lui-même

> 💡 *En C#, si ton code contient une erreur de syntaxe ou de type, le programme **refuse de compiler**. Tu es forcé de corriger avant même de lancer quoi que ce soit — ce qui est une bonne habitude.*

À retenir

- C# est un langage créé par **Microsoft**, utilisé notamment pour les jeux (Unity) et les apps Windows
- C'est un langage **compilé** : le code est transformé en fichier exécutable avant d'être lancé
- Il compile vers un **code intermédiaire** exécuté par le **.NET CLR**

- Les erreurs sont détectées **à la compilation** — avant l'exécution

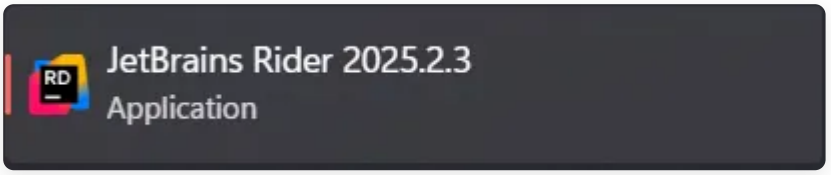
Mise en place de l'environnement C#

> *Avant d'écrire du code, on installe les bons outils et on comprend comment ils s'organisent. Cette étape ne prend que quelques minutes et servira pour toute la suite du cours.*

Les outils nécessaires

Pour développer en C# sous Windows, on a besoin de deux choses :

OUTIL	RÔLE
.NET SDK	La "machinerie" qui compile et exécute le code C#
JetBrains Rider	L'éditeur intelligent (IDE) dans lequel on code



Il existe d'autres outils pour programmer en C# (Visual Studio, VS Code...). Je recommande Rider pour ce cours.

Installer Rider

Licence gratuite pour les étudiants

JetBrains propose une licence éducation **entièrement gratuite** pour les élèves et enseignants.

👉 Crée ton compte sur jetbrains.com/student avec ton adresse mail scolaire.

Téléchargement

👉 jetbrains.com/rider → **Windows** → `Download`

Lance l'installateur et accepte les options par défaut.

Le .NET SDK

Le SDK (*Software Development Kit*) est l'ensemble des outils qui permettent de **compiler** et **exécuter** du code C#.

> 💡 ***Bonne nouvelle*** *: tu n'as pas besoin de l'installer séparément à l'avance.*
> *Quand tu crées un nouveau projet dans Rider, il détecte automatiquement si le SDK requis est présent — et te propose de le* ***télécharger directement*** *depuis l'interface si ce n'est pas le cas.*

Si tu préfères l'installer manuellement :
👉 dotnet.microsoft.com/download → choisis la version **LTS** (Long Term Support)

Pour vérifier qu'il est bien installé, ouvre un terminal et tape :

```
dotnet --version
```

Tu dois voir quelque chose comme `8.0.x`.

Solution et Projet : comment ça s'organise ?

Avant de créer quoi que ce soit, il faut comprendre deux concepts clés dans Rider (et dans tout l'écosystème .NET) :

La Solution

Une **Solution** est le **conteneur global** de ton travail. Elle regroupe un ou plusieurs projets.

Concrètement, c'est un fichier `.sln` qui sert de point d'entrée à Rider.

Le Projet

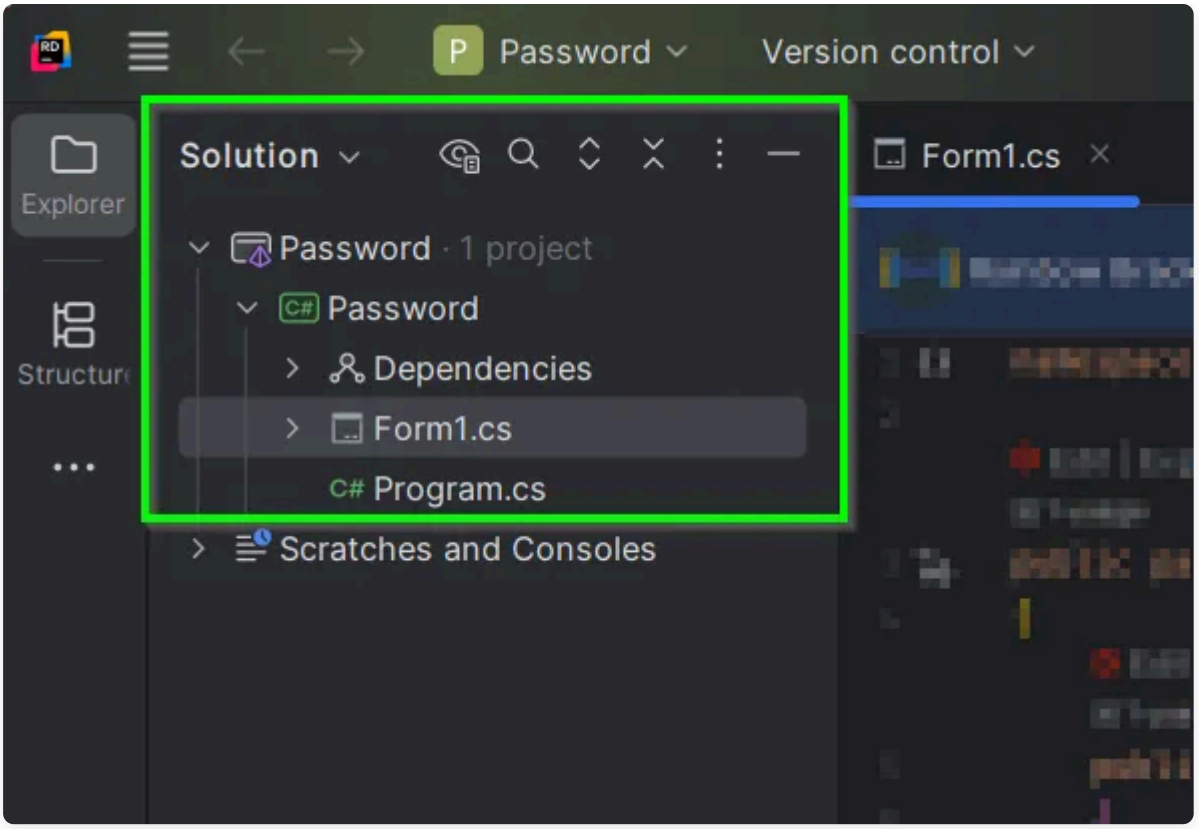
Un **Projet** est une **application autonome** à l'intérieur de la solution — avec son propre code, ses propres fichiers, et un fichier `.csproj` qui le décrit.

```
MaSolution/              ← Solution (.sln)
│
├── MonApplication/      ← Projet (.csproj)
│   └── Program.cs
│
└── AutreApplication/    ← 2e Projet (si besoin)
    └── Program.cs
```

En cours, on aura souvent **une solution = un projet**. Mais dès qu'on aborde la POO, on verra qu'il peut être utile de séparer les choses.

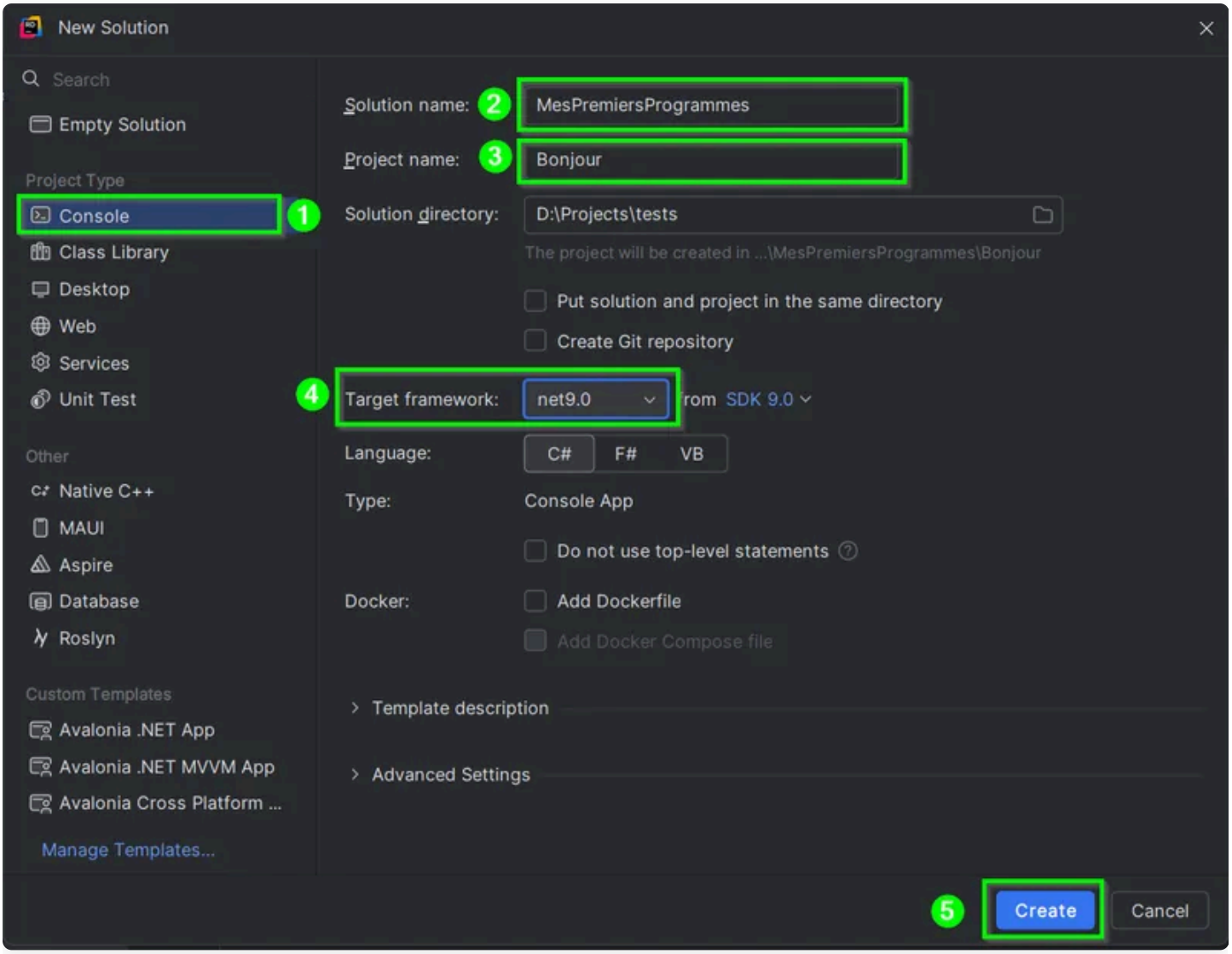
Exemple

Tu as ici une solution "Password" qui contient un projet "Password", qui contien à son tours des fichiers c-sharp:


Password
Version control
Explorer
Solution
Form1.cs
Password · 1 project
Password
Dependencies
Form1.cs
Program.cs
Scratches and Consoles

Créer un premier projet

Étape par étape



- Lance Rider → clique sur **New Solution**
- Dans la liste de gauche, choisis **Console Application [1]**
- Remplis les champs :
  - **Solution name** : `MesPremiersProgrammes` (pas d'espaces, pas d'accents) **[2]**
  - **Project name** : `Bonjour` **[3]**
  - **Location** : ton dossier de travail
  - **Target Framework** : choisis la version **LTS** disponible (ex: `.NET 9`) **[4]**

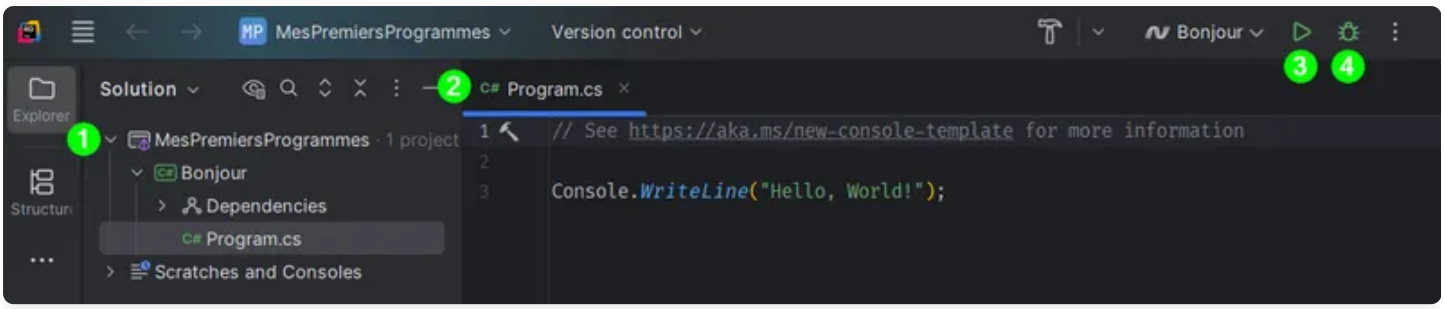
> 💡 *Si le SDK n'est pas installé, Rider affiche un avertissement à cette étape avec un bouton **Download .NET SDK**. Clique dessus — il s'installe automatiquement.*

- Clique sur **Create [5]**

Comprendre l'arborescence générée

Une fois le projet créé:

Rider affiche ceci:



Sur ton disque, tu auras la structure suivante:




```
MesPremiersProgrammes [1]
  |
  ├── Bonjour/
  |     |
  |     ├── Program.cs            ← TON code est ici [2]
  |     |
  |     ├── bin/                  ← fichiers compilés (générés automatiquement, ne pas toucher)
  |     └── obj/                  ← fichiers temporaires (générés automatiquement, ne pas
toucher)
  └──
```

Le seul fichier que tu modifies au quotidien : `Program.cs`.

Exécuter un programme

Rider propose deux façons de lancer ton programme :

ACTION	RACCOURCI	QUAND L'UTILISER
Exécuter [3]	`Shift + F10`	Pour voir le résultat
Déboguer [4]	`Shift + F9`	Pour inspecter pas à pas

La sortie apparaît dans le panneau **Run** en bas de l'écran.

Le débogueur : ton meilleur allié

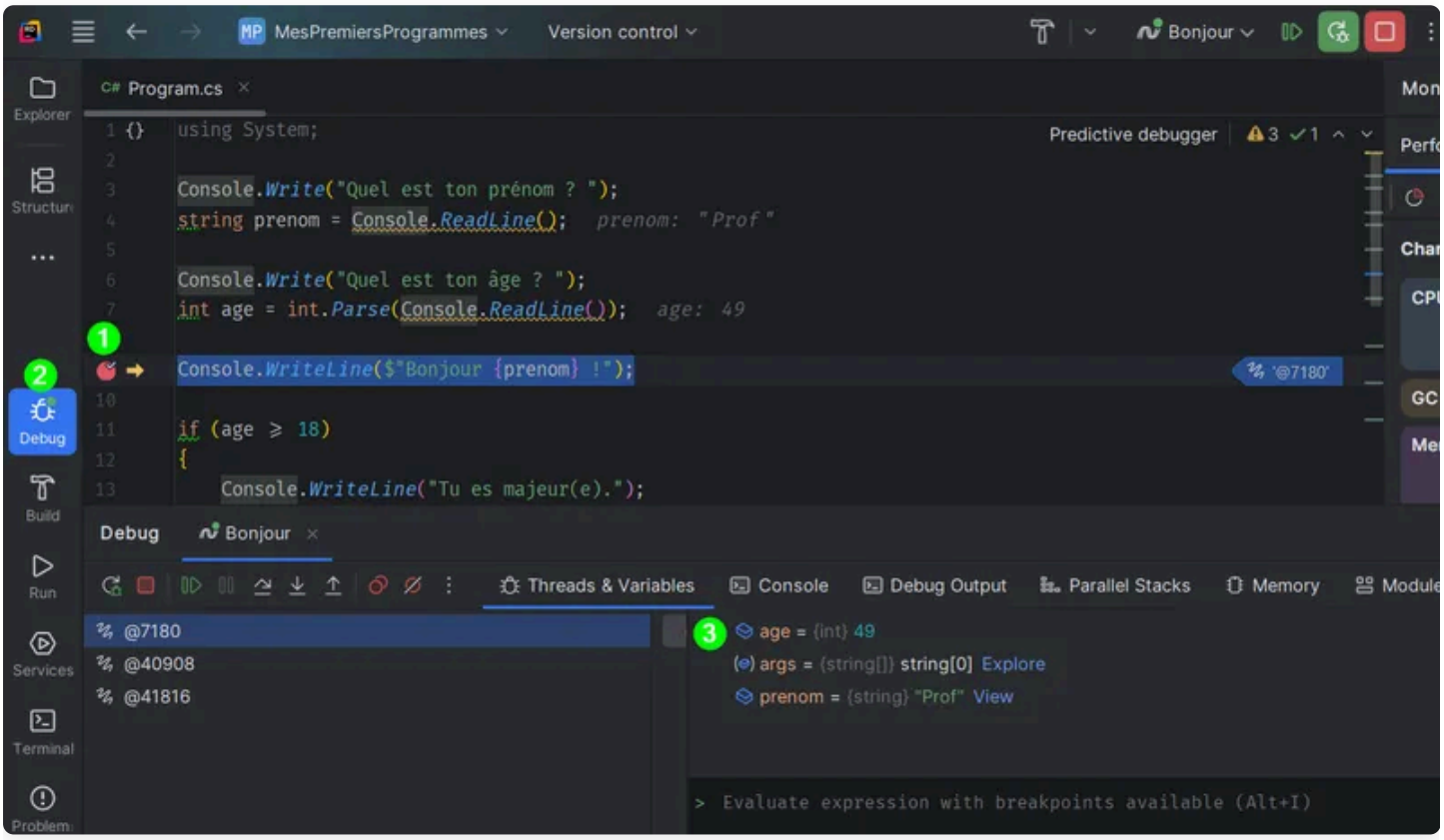
Le débogueur permet de **mettre le programme en pause** à une ligne précise pour inspecter les valeurs des variables.

▸ Poser un breakpoint

Clique dans la marge à gauche d'une ligne → un point rouge 🔴 apparaît.



Lance ensuite en mode **Debug** (`Shift + F9`).



Le programme s'arrête à cette ligne. Tu peux voir dans le panneau **Debug [2]** la valeur de chaque variable à cet instant.

▸ Avancer pas à pas

RACCOURCI	ACTION
`F8`	Exécuter la ligne et passer à la suivante
`F7`	Entrer dans une méthode
`F9`	Continuer jusqu'au prochain breakpoint

Raccourcis essentiels

À connaître dès le premier jour :

RACCOURCI	ACTION
`Shift + F10`	Exécuter
`Shift + F9`	Déboguer
`F9`	Poser / enlever un breakpoint
`F8`	Pas à pas
`Ctrl + /`	Commenter / décommenter une ligne
`Alt + Entrée`	Correction / suggestion automatique
`Ctrl + Z`	Annuler
`Ctrl + Shift + Z`	Rétablir

> 💡 `Alt + Entrée` *est probablement le raccourci le plus puissant de Rider. Quand tu ne sais pas quoi faire face à une erreur, essaie-le.*

🧪 Exercice — Vérifier que tout fonctionne

Objectif : confirmer que Rider et .NET sont correctement installés.

Consigne : dans `Program.cs`, écris ce code et exécute-le :

```
Console.WriteLine("Rider est prêt !");
Console.WriteLine($"Version .NET : {Environment.Version}");
```

Résultat attendu :

```
Rider est prêt !
Version .NET : 8.0.x
```

Si tu vois ces deux lignes dans le panneau Run : **ton environnement est opérationnel** ✅

À retenir

- Le **.NET SDK** compile et exécute le code — Rider peut le télécharger pour toi
- **Rider** est l'IDE — licence gratuite avec ton adresse scolaire
- Une **Solution** contient un ou plusieurs **Projets**
- `Program.cs` est le fichier où tu codes
- `bin/` et `obj/` sont générés automatiquement — ne pas y toucher
- `Shift + F10` pour exécuter, `Shift + F9` pour déboguer

Alternative — Visual Studio Code

Si Rider n'est pas disponible sur ta machine, tu peux coder en C# avec **Visual Studio Code**, éditeur gratuit et léger de Microsoft.

Installation

1. 👉 code.visualstudio.com → télécharge et installe VS Code
2. Dans VS Code, ouvre le panneau Extensions (`Ctrl + Shift + X`)
3. Recherche et installe **C# Dev Kit** (éditeur : Microsoft)

Le .NET SDK reste nécessaire — installe-le depuis dotnet.microsoft.com/download si ce n'est pas déjà fait.

Ce que tu y trouveras

- Coloration syntaxique et autocomplétion
- Exécution et débogage intégrés
- Gestion de la Solution et du Projet via un panneau dédié

Ce que tu n'y trouveras pas

VS Code reste moins confortable que Rider pour le C# : le refactoring automatique, le débogueur et les suggestions intelligentes sont nettement moins puissants. C'est une solution de dépannage, pas un remplacement.

> 💡 *Si tu connais déjà VS Code pour PHP ou JavaScript, tu t'y sentiras à l'aise — mais investis du temps pour configurer Rider dès que possible.*

La syntaxe de base du C#

> *Dans cet article, on présente les bases essentielles du langage C# dans un ordre logique, pour comprendre comment écrire et organiser un programme simple.*

Ce qu'on va savoir faire

À la fin de cet article, tu seras capable d'écrire ce programme complet :

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("Quel est ton prénom ? ");
        string prenom = Console.ReadLine();

        Console.Write("Quel est ton âge ? ");
        int age = int.Parse(Console.ReadLine());

        Console.WriteLine($"Bonjour {prenom} !");

        if (age >= 18)
        {
            Console.WriteLine("Tu es majeur(e).");
        }
        else
        {
            Console.WriteLine("Tu es mineur(e).");
        }

        Console.WriteLine("Comptons jusqu'à ton âge :");
        for (int i = 1; i <= age; i++)
        {
            Console.WriteLine(i);
        }
    }
}
```

Le bloc de base

En C# moderne, tu peux écrire ton code **directement**, sans structure autour :

```
// Mon premier programme
Console.WriteLine("Bonjour !");
```

C'est suffisant. Le compilateur s'occupe du reste dans les coulisses. C'est ce qu'on appelle des ***top-level statements***.

> 💡 *Si tu croises cette structure dans des tutoriels ou anciens projets, ne sois pas surpris(e) :*
>
> ```
> using System;
>
> class Program
> {
>     static void Main()
>     {
>         Console.WriteLine("Bonjour !");
>     }
> }
> ```

> *C'est l'ancienne façon d'écrire — obligatoire avant 2020. Elle est toujours valide, et on la retrouvera naturellement quand on abordera la programmation orientée objet.*

Nuance importante à connaître

Un seul fichier du projet peut utiliser les top-level statements. Dès qu'on a plusieurs fichiers/classes, la structure complète redevient nécessaire.

🧪 Exercice 1 — Premier programme

Objectif : créer et exécuter un programme minimal.

Consigne : écris un programme qui affiche ces deux lignes :---

🧪 Exercice 1 — Premier programme

Objectif : créer et exécuter un programme minimal.

Consigne : recopie la structure de base et ajoute cette ligne dans `Main()` :

```
Console.WriteLine("Mon premier programme C# !");
```

Résultat attendu :

```
Mon premier programme C# !
```

À retenir : `Console.WriteLine()` affiche une ligne dans la console.

---

2. Variables

Une **variable** sert à stocker une valeur pour la réutiliser.

```
string prenom = "Alice";
int age = 17;
```

Syntaxe :

```
type nom = valeur;
```

Les types les plus courants pour commencer :

TYPE	CONTIENT	EXEMPLE
`int`	un entier	`42`
`double`	un décimal	`3.14`
`string`	du texte	`"Bonjour"`
`bool`	vrai ou faux	`true / false`

⚠️ *Un `string` se note avec des guillemets `"..."`, un `char` avec des apostrophes `'...'`.*

🧪 Exercice 2 — Déclarer des variables

Objectif : déclarer et afficher plusieurs variables.

Consigne : déclare ces trois variables, puis affiche-les chacune sur une ligne séparée :

```
Prénom : Alice
Âge    : 17
Taille : 1.68
```

Hint : utilise `Console.WriteLine()` trois fois.

---

3. Afficher avec interpolation

La façon la plus lisible d'afficher du texte **et** des variables ensemble :

```
string prenom = "Alice";
int age = 17;

Console.WriteLine($"Bonjour {prenom}, tu as {age} ans.");
```

Résultat :

```
Bonjour Alice, tu as 17 ans.
```

> 💡 *Le `$` devant les guillemets active l'interpolation. Les variables se placent entre `{ }`.*

🪛 Exercice 3 — Interpolation

Objectif : utiliser l'interpolation de chaînes.

Consigne : à partir des variables `prenom`, `age` et `taille` de l'exercice précédent, affiche **une seule ligne** :

```
Alice a 17 ans et mesure 1.68 m.
```

---

4. Lire une entrée utilisateur

`Console.ReadLine()` lit ce que l'utilisateur tape :

```
Console.Write("Quel est ton prénom ? ");
string prenom = Console.ReadLine();
```

> 💡 *`Console.Write()` n'ajoute pas de retour à la ligne — pratique pour mettre la réponse sur la même ligne que la question.*

Convertir l'entrée

`ReadLine()` renvoie **toujours du texte**. Si tu attends un nombre, il faut convertir :

```
Console.Write("Quel est ton âge ? ");
int age = int.Parse(Console.ReadLine());
```

▸ 🧪 Exercice 4 — Saisie utilisateur

Objectif : lire des données saisies et les réafficher.

Consigne : écris un programme qui demande le prénom et l'année de naissance, puis affiche :

```
Bonjour Alice ! Tu es né(e) en 2007.
```

---

5. Opérations sur les nombres

```
int a = 10;
int b = 3;

Console.WriteLine(a + b);   // 13
Console.WriteLine(a - b);   // 7
Console.WriteLine(a * b);   // 30
Console.WriteLine(a / b);   // 3  (division entière !)
Console.WriteLine(a % b);   // 1  (reste de la division)
```

> ⚠️ *La division entre deux `int` donne un `int` : `10 / 3 = 3`, pas `3.33`.*
> *Pour un résultat décimal, utilise `double` : `10.0 / 3 = 3.333...`*

Raccourcis utiles :

```
x += 5;   // x = x + 5
x -= 2;
x *= 3;
x /= 2;
x++;      // x = x + 1
x--;      // x = x - 1
```

▸ 🧪 Exercice 5 — Calculatrice simple

Objectif : combiner saisie, conversion et opérations.

Consigne : demande deux entiers à l'utilisateur, puis affiche :

```
Somme      : 13
Différence : 7
Produit    : 30
Reste      : 1
```

---

6. Les conditions

▸ if / else

```
if (age >= 18)
{
    Console.WriteLine("Majeur");
}
else
{
    Console.WriteLine("Mineur");
}
```

▸ if / else if / else

```
if (age < 12)
{
    Console.WriteLine("Enfant");
}
else if (age < 18)
{
    Console.WriteLine("Adolescent");
}
else
{
    Console.WriteLine("Adulte");
}
```

Opérateurs de comparaison :

OPÉRATEUR	SIGNIFICATION
`==`	égal à
`≠`	différent de
`>`	strictement plus grand
`<`	strictement plus petit
`≥`	plus grand ou égal
`≤`	plus petit ou égal

▸ 🧪 Exercice 6 — Conditions

Objectif : utiliser `if / else if / else`.

Consigne : demande une note sur 20, puis affiche :

```
18 → Excellent !
14 → Bien
10 → Suffisant
8  → Insuffisant
```

Définis toi-même les seuils.

---

7. Les boucles

Boucle `while`

Répète **tant que** la condition est vraie :

```
int i = 0;

while (i < 5)
{
    Console.WriteLine(i);
    i++;
}
```

Boucle `for`

Quand on connaît le nombre de répétitions à l'avance :

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Structure d'un `for` :

```
for ( initialisation ; condition ; mise à jour )
```

🧪 Exercice 7 — Boucles

Objectif : utiliser une boucle `for`.

Consigne : demande un nombre `n` à l'utilisateur et affiche sa **table de multiplication** de 1 à 10 :

```
3 x 1  =  3
3 x 2  =  6
3 x 3  =  9
...
3 x 10 = 30
```

---

8. Exercices de synthèse

Ces exercices combinent plusieurs notions.

▸ 🏁 Synthèse 1 — Le programme d'accueil

Reproduis le programme présenté en introduction (sans regarder !).
Il doit demander prénom + âge, afficher un message personnalisé, indiquer si l'utilisateur est majeur ou non, et compter jusqu'à son âge.

▸ 🏁 Synthèse 2 — Pair ou impair

Consigne : demande 5 nombres à l'utilisateur (avec une boucle `for`), et pour chacun indique s'il est pair ou impair.

Hint : `nombre % 2 == 0` → pair.

Résultat attendu :

```
Nombre 1 : 7 → impair
Nombre 2 : 4 → pair
...
```

▸ 🏁 Synthèse 3 — Devine le nombre

Consigne : le programme choisit le nombre 42. L'utilisateur doit le deviner.
Tant qu'il n'a pas trouvé, le programme dit *"trop grand"* ou *"trop petit"*.
Quand il trouve, il affiche le nombre d'essais.

Hint : utilise une boucle `while` et un compteur.

Annexe — Référence rapide

```
// Afficher
Console.WriteLine("texte");          // avec retour à la ligne
Console.Write("texte");              // sans retour à la ligne
Console.WriteLine($"Bonjour {nom}"); // avec variable

// Lire
string texte = Console.ReadLine();

// Convertir
int n    = int.Parse(Console.ReadLine());
double d = double.Parse(Console.ReadLine());

// Commentaires
// ligne unique
/* plusieurs
   lignes */
```

Exercice — Todo List en console

> *Un exercice de synthèse pour pratiquer les variables, les boucles, les conditions et les tableaux dynamiques — en construisant une vraie mini-application interactive.*

Objectif

Créer une **application console interactive** de gestion de tâches (Todo List).

L'utilisateur navigue dans un menu, peut ajouter des tâches, les afficher, les marquer comme faites, et en supprimer.

Aucune classe n'est utilisée — uniquement des `List<string>`, des boucles et des conditions.

Résultat attendu

Écran d'accueil

```
╔═══════════════════════════╗
║       MA TODO LIST        ║
╚═══════════════════════════╝

 1. Voir les tâches
 2. Ajouter une tâche
 3. Marquer comme faite
 4. Supprimer une tâche
 0. Quitter

Votre choix :
```

Option 1 — Voir les tâches (liste vide)

```
--- Mes tâches ---
(aucune tâche pour l'instant)
```

Option 1 — Voir les tâches (avec tâches)

```
--- Mes tâches ---
 1. [ ] Faire les courses
 2. [ ] Réviser C#
 3. [✓] Promener le chien
```

> `[ ]` = *à faire*
> `[✓]` = *terminée*

Option 2 — Ajouter une tâche

```
Nouvelle tâche : Faire les courses
✅ Tâche ajoutée !
```

Option 3 — Marquer comme faite

```
--- Mes tâches ---
 1. [ ] Faire les courses
 2. [ ] Réviser C#

Numéro de la tâche à cocher : 1
✅ Tâche marquée comme faite !
```

Option 4 — Supprimer une tâche

```
--- Mes tâches ---
 1. [✓] Faire les courses
 2. [ ] Réviser C#

Numéro de la tâche à supprimer : 1
🗑️  Tâche supprimée.
```

Option 0 — Quitter

```
À bientôt !
```

▸ **Erreur de saisie**

```
Votre choix : 9
⚠️  Choix invalide. Réessayez.
```

Contraintes

- Le menu se **réaffiche automatiquement** après chaque action
- Si la liste est vide et que l'utilisateur choisit l'option 3 ou 4, afficher :

  ```
  ⚠️  Aucune tâche à modifier.
  ```

- Si l'utilisateur entre un numéro de tâche inexistant (option 3 ou 4) :

  ```
  ⚠️  Numéro invalide.
  ```

- Aucune classe autorisée — tout dans `Program.cs`

Données à utiliser

Tu auras besoin de **deux listes parallèles** :

```
List<string> taches = new List<string>();
List<bool> faites = new List<bool>();
```

- `taches[i]` contient le texte de la tâche numéro `i`
- `faites[i]` vaut `true` si la tâche est cochée, `false` sinon

Décomposition suggérée

Travaille dans cet ordre — teste après chaque étape :

Étape 1 — Afficher le menu en boucle (`while` infini + `break` sur `0`)

Étape 2 — Lire le choix et afficher un message selon l'option choisie (juste un `Console.WriteLine` de test)

Étape 3 — Implémenter "Ajouter une tâche"

Étape 4 — Implémenter "Voir les tâches" (avec `[ ]` / `[✓]`)

Étape 5 — Implémenter "Marquer comme faite"

Étape 6 — Implémenter "Supprimer une tâche"

Notions mobilisées

NOTION	UTILISÉE POUR
`List<string>`	stocker les tâches
`List<bool>`	mémoriser l'état de chaque tâche
`while`	boucle principale du menu
`switch`	aiguiller selon le choix
`for`	afficher et parcourir les tâches
`int.Parse`	convertir le choix et le numéro
interpolation `$""`	afficher `[ ]` ou `[✓]` dynamiquement

- **Étape bonus possible** : "sauvegarder dans un fichier `.txt` " si tu veux introduire `File.WriteAllLines` plus tard

Exercice — Générateur de mot de passe (app fenêtrée)

> *Premier vrai projet Windows : tu vas construire une fenêtre avec des boutons, des cases à cocher, un champ de texte, et y connecter du code C# pour générer un mot de passe selon les choix de l'utilisateur. L'objectif n'est pas seulement de produire l'app — c'est de comprendre comment une interface graphique et du code se parlent.*

C'est un projet **ambitieux** : tu vas écrire ta première application Windows native, avec une vraie fenêtre, des contrôles cliquables et du code qui réagit à l'utilisateur. Ne te laisse pas impressionner par la longueur de l'énoncé — chaque étape est petite et l'ordre est pensé pour que rien ne te bloque.

L'application à construire s'appelle **Générateur de mot de passe**. L'utilisateur choisit la longueur souhaitée, coche les types de caractères qu'il veut inclure (minuscules, majuscules, chiffres, caractères spéciaux), clique sur un bouton, et un mot de passe aléatoire s'affiche.

🎯 Objectifs d'apprentissage

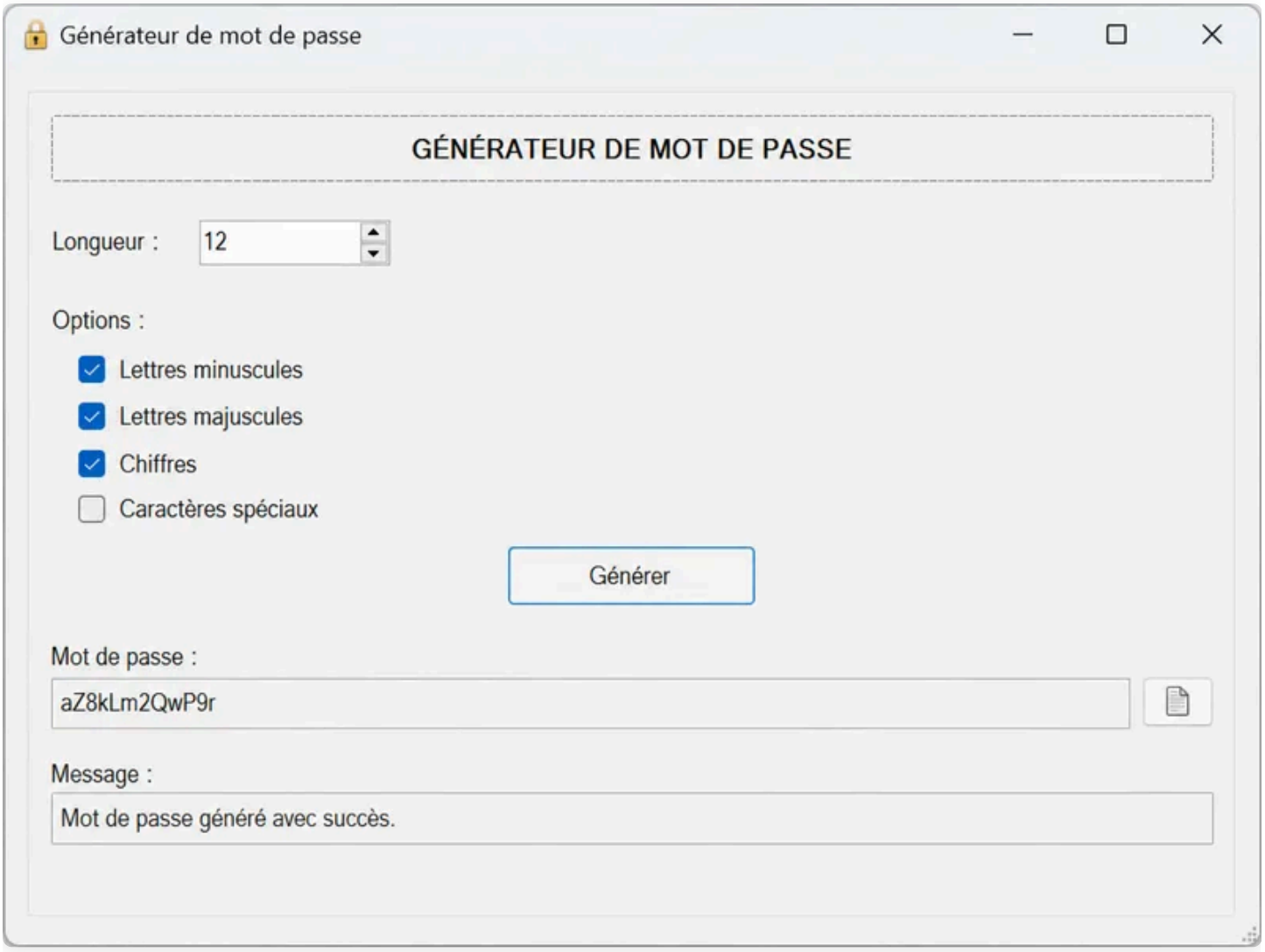
À la fin de ce projet, tu seras capable de :

1. Créer une application **Windows Forms** dans JetBrains Rider
2. Construire une interface en plaçant des composants graphiques (`Label` , `Button` , `CheckBox` , etc.)
3. Renommer correctement tes composants et comprendre pourquoi c'est essentiel
4. Connecter un **événement** (clic sur un bouton) à une **méthode** de ton code
5. Lire les valeurs saisies par l'utilisateur et les utiliser dans une logique
6. Utiliser la classe `Random` pour générer un résultat aléatoire
7. Construire dynamiquement une chaîne de caractères
8. Gérer les cas d'erreur de base et afficher un message clair

---

🖼️ Aperçu de l'interface

Voici à quoi ressemblera ton application une fois terminée :



Six éléments à placer : un titre, un champ pour la longueur, quatre cases à cocher, un bouton **Générer**, une zone de texte pour le résultat, et un message d'information.

🧠 Avant de commencer : ce que tu dois savoir

Windows Forms, c'est quoi ?

Windows Forms (souvent abrégé *WinForms*) est la technologie historique de Microsoft pour créer des applications fenêtrées Windows. Elle existe depuis 2002. C'est la plus simple à prendre en main parce qu'elle permet de **dessiner ton interface à la souris** dans un éditeur visuel — pas besoin d'écrire du XML ni du code pour positionner les contrôles.

Une application WinForms est composée de :

- **Une ou plusieurs** `Form` — chaque fenêtre est une classe `Form`
- **Des contrôles** posés sur la fenêtre (`Button` , `Label` , `TextBox` …)

- **Du code C#** qui réagit aux actions de l'utilisateur

Le modèle événementiel

C'est le concept central à comprendre. Contrairement à un programme console qui s'exécute ligne par ligne du début à la fin, une application fenêtrée :

1. Affiche la fenêtre
2. **Attend** que l'utilisateur fasse quelque chose (clic, frappe au clavier…)
3. Quand quelque chose arrive, exécute la méthode **liée** à cette action
4. Retourne attendre

On dit que l'application est **pilotée par les événements** (GB ***event-driven***). Cliquer sur le bouton **Générer** déclenchera l'**événement** `Click`, qui appellera la **méthode** que tu auras écrite.

> 💡 *Cette façon de penser est très différente d'un programme console. Au début, l'idée de "code qui ne s'exécute qu'à la demande" peut surprendre. C'est exactement comme une application mobile : rien ne se passe tant que tu ne touches pas l'écran.*

Étape 1 — Créer la solution dans Rider

Procédure

1. Ouvre **JetBrains Rider**
2. Clique sur **New Solution**
3. Dans la liste de gauche, choisis : **.NET** → **Desktop** → **Windows Forms App**

Configuration

Remplis les champs comme ceci :

CHAMP	VALEUR
Solution name	`GenerateurMotDePasse`
Project name	`GenerateurMotDePasseApp`
Solution directory	un dossier clair (par exemple `D:\Projets\C#\`)
Framework	`.NET 8.0` (ou la version utilisée en classe)

> 💡 ***Solution vs Projet*** *: une solution est un conteneur qui peut regrouper plusieurs projets (par exemple : une UI + une bibliothèque + des tests). Ici, on n'a qu'un seul projet, mais Rider impose cette structure — c'est normal.*

Vérification

Clique sur le bouton **Run** (▶ vert, en haut). Une fenêtre vide s'affiche. Ferme-la.

👉 Si tu vois cette fenêtre vide, **tout fonctionne**. Tu peux passer à l'étape suivante. Sinon, retourne aux étapes de configuration.

Étape 2 — Construire l'interface

Dans l'explorateur de fichiers de Rider, double-clique sur `Form1.cs`. Une vue spéciale s'ouvre : c'est le **concepteur visuel** (GB ***designer***) qui te permet de dessiner ta fenêtre.

La boîte à outils

À droite (ou via le menu **View** → **Toolbox**), tu trouves la **boîte à outils** (GB ***Toolbox***) avec tous les contrôles disponibles. Tu vas y piocher.

Placer les contrôles

Glisse-dépose les contrôles suivants sur ta `Form1` :

CONTRÔLE	RÔLE DANS L'APP
`Label`	Le **titre** de la fenêtre
`Label`	Texte « Longueur : »
`NumericUpDown`	Le champ numérique pour la longueur
`CheckBox` × 4	Une case par type de caractère
`Button`	Le bouton **Générer**
`Label`	Texte « Mot de passe : »
`TextBox`	La zone qui affichera le résultat
`Label`	La ligne de message

Place-les en respectant grossièrement le mockup plus haut. La perfection visuelle peut attendre — on s'en occupe à la fin.

Renommer les contrôles : indispensable

Sélectionne chaque contrôle, puis dans le panneau **Properties** (à droite), trouve la propriété `(Name)` et change-la. **Sans renommage clair, tu te perdras dans ton propre code dans 5 minutes.**

CONTRÔLE	(NAME)	AUTRE PROPRIÉTÉ À RÉGLER
Label titre	`lblTitre`	`Text = "GÉNÉRATEUR DE MOT DE PASSE"`
NumericUpDown	`numLongueur`	`Minimum = 4`, `Maximum = 64`, `Value = 12`
CheckBox 1	`chkMinuscules`	`Text = "Lettres minuscules"`, `Checked = true`
CheckBox 2	`chkMajuscules`	`Text = "Lettres majuscules"`, `Checked = true`
CheckBox 3	`chkChiffres`	`Text = "Chiffres"`, `Checked = true`
CheckBox 4	`chkSpeciaux`	`Text = "Caractères spéciaux"`
Button	`btnGenerer`	`Text = "Générer"`
TextBox	`txtResultat`	`ReadOnly = true`, `Font` plus grosse
Label message	`lblMessage`	`Text = ""` (vide au départ)

▸ Les conventions de nommage

Tu as remarqué les préfixes : `lbl` pour Label, `txt` pour TextBox, `btn` pour Button, `chk` pour CheckBox, `num` pour NumericUpDown. Ce n'est pas obligatoire pour le compilateur — c'est une **convention** qui te permet de voir immédiatement de quel type est un contrôle juste en lisant son nom dans le code.

> 💡 *Sans convention, tu te retrouves avec des `password1`, `password2`, `password3` et tu ne sais plus lequel est le champ texte, lequel est le label, lequel est la case. Avec convention : `txtPassword`, `lblPassword`, `chkPassword` — zéro ambiguïté.*

Étape 3 — Connecter le bouton à du code (test)

Avant d'écrire la vraie logique, on fait un test minimal pour s'assurer que le bouton réagit.

Double-clique sur le bouton `btnGenerer` dans le concepteur. Rider :

1. Crée automatiquement une méthode appelée `btnGenerer_Click` dans `Form1.cs`
2. Lie cette méthode à l'événement `Click` du bouton
3. T'ouvre le code à cet endroit précis

Tu vois quelque chose comme :

```
private void btnGenerer_Click(object sender, EventArgs e)
{

}
```

Tape dedans :

```
private void btnGenerer_Click(object sender, EventArgs e)
{
    MessageBox.Show("Bouton cliqué !");
}
```

Lance l'application (▶). Clique sur **Générer**. Une boîte de dialogue doit apparaître.

> 💡 ***Pourquoi ce test ?*** *Parce qu'il **isole** une seule responsabilité : le bouton réagit-il ? Si oui, on sait que tout le câblage interface↔code fonctionne. Si plus tard ta vraie logique ne marche pas, tu sauras que le problème est dans la logique, pas dans la connexion. C'est une habitude à prendre dans tous tes projets : avancer par petits pas validés.*

Étape 4 — Lire les valeurs saisies par l'utilisateur

Remplace le `MessageBox.Show` par le code suivant :

```
private void btnGenerer_Click(object sender, EventArgs e)
{
    int longueur = (int)numLongueur.Value;

    bool inclureMinuscules = chkMinuscules.Checked;
    bool inclureMajuscules = chkMajuscules.Checked;
    bool inclureChiffres   = chkChiffres.Checked;
    bool inclureSpeciaux   = chkSpeciaux.Checked;
}
```

▸ Décortiquons

- `numLongueur.Value` retourne un `decimal` (pour pouvoir gérer aussi des valeurs à virgule). On le **convertit** explicitement en `int` avec `(int)…` — c'est un **cast**.
- `chkMinuscules.Checked` est un `bool` qui vaut `true` si la case est cochée, `false` sinon. Pas besoin de conversion.

> 💡 ***Conseil de méthode*** *: à ce stade, tu peux temporairement ajouter* `MessageBox.Show($"Longueur : {longueur}");` *à la fin de la méthode pour vérifier que la valeur est bien lue. C'est une forme rudimentaire de débogage. Tu retireras la ligne avant de continuer.*

Étape 5 — Construire l'alphabet de caractères autorisés

L'idée : selon les cases cochées, on assemble une **chaîne** qui contiendra tous les caractères qu'on autorise à tirer au sort.

```
string caracteres = "";

if (inclureMinuscules) caracteres += "abcdefghijklmnopqrstuvwxyz";
if (inclureMajuscules) caracteres += "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
if (inclureChiffres)   caracteres += "0123456789";
if (inclureSpeciaux)   caracteres += "!@#$%&*?";
```

Après ce bloc, la variable `caracteres` contient **uniquement** les caractères que l'utilisateur a autorisés. Si toutes les cases sont cochées, la chaîne fait 71 caractères. Si une seule case est cochée, elle est plus courte. Si aucune case n'est cochée, elle est **vide** — et c'est un problème qu'on doit gérer.

Étape 6 — Refuser les cas absurdes

Si la chaîne `caracteres` est vide, impossible de générer un mot de passe. Il faut prévenir l'utilisateur **avant** d'essayer.

```
if (caracteres == "")
{
    MessageBox.Show("Sélectionne au moins un type de caractère.",
                    "Aucune option choisie",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Warning);
    lblMessage.Text = "Erreur : aucune option cochée.";
    return;
}
```

Le `return` est important : il **interrompt** la méthode. Sans lui, le code continuerait et planterait à l'étape suivante en essayant de lire un caractère dans une chaîne vide.

> 💡 ***Principe à retenir*** *: valide toujours tes entrées **avant** de t'en servir. Plus tôt tu détectes une situation impossible, plus le message d'erreur est clair pour l'utilisateur.*

Étape 7 — Générer le mot de passe

C'est le cœur de la logique. On va tirer aléatoirement, `longueur` fois, un caractère dans la chaîne `caracteres`, et le concaténer au résultat.

```
Random rnd = new Random();
string motDePasse = "";

for (int i = 0; i < longueur; i++)
{
    int index = rnd.Next(caracteres.Length);
    motDePasse += caracteres[index];
}
```

Décortiquons

- `new Random()` crée un générateur de nombres aléatoires
- `rnd.Next(caracteres.Length)` retourne un entier entre `0` (inclus) et `caracteres.Length` (exclu) — exactement la plage des indices valides de la chaîne
- `caracteres[index]` extrait le caractère à cet indice
- `motDePasse += …` ajoute ce caractère à la fin du résultat
- La boucle `for` répète l'opération `longueur` fois

> ⚠️ ***Remarque importante (sécurité réelle)*** *:* `Random` *est pseudo-aléatoire, pas cryptographiquement sûr. Pour ce projet pédagogique, c'est suffisant. Pour une vraie application qui génère de* ***vrais*** *mots de passe à utiliser, on utiliserait* `RandomNumberGenerator` *de l'espace de noms* `System.Security.Cryptography`*. À garder en tête pour la suite.*

Étape 8 — Afficher le résultat

```
txtResultat.Text = motDePasse;
lblMessage.Text  = $"Mot de passe de {longueur} caractères généré.";
```

L'interpolation `$"…"` permet d'insérer la valeur de `longueur` directement dans la chaîne. C'est l'équivalent moderne et lisible de la concaténation `"… " + longueur + " …"`.

Le code complet de l'événement

Récapitulons tout en un seul bloc — c'est ce que tu dois avoir dans `Form1.cs` :

```
private void btnGenerer_Click(object sender, EventArgs e)
{
    int longueur = (int)numLongueur.Value;

    string caracteres = "";
    if (chkMinuscules.Checked) caracteres += "abcdefghijklmnopqrstuvwxyz";
    if (chkMajuscules.Checked) caracteres += "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    if (chkChiffres.Checked)   caracteres += "0123456789";
    if (chkSpeciaux.Checked)   caracteres += "!@#$%&*?";

    if (caracteres == "")
    {
        MessageBox.Show("Sélectionne au moins un type de caractère.",
                        "Aucune option choisie",
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Warning);
        lblMessage.Text = "Erreur : aucune option cochée.";
        return;
    }

    Random rnd = new Random();
    string motDePasse = "";

    for (int i = 0; i < longueur; i++)
    {
        int index = rnd.Next(caracteres.Length);
        motDePasse += caracteres[index];
    }

    txtResultat.Text = motDePasse;
    lblMessage.Text  = $"Mot de passe de {longueur} caractères généré.";
}
```

Lance ton application. Coche les options, choisis une longueur, clique **Générer**. Tu dois voir un mot de passe différent à chaque clic.

🧪 Tests à effectuer

Vérifie systématiquement les cas suivants avant de considérer ton app comme terminée :

TEST	RÉSULTAT ATTENDU
Toutes cases cochées, longueur 12	Mot de passe mixte de 12 caractères
Seulement chiffres, longueur 6	Mot de passe de 6 chiffres
Aucune case cochée, clic	Message d'erreur, pas de plantage
Longueur 4	Mot de passe court généré
Longueur 64	Mot de passe long généré
Deux clics consécutifs	Deux mots de passe différents

📝 Version minimale obligatoire

Pour valider le projet, ton application doit :

- générer un mot de passe au clic
- respecter les choix de l'utilisateur (longueur + types de caractères)
- afficher le résultat dans la zone prévue
- gérer **au moins** le cas « aucune case cochée »

🌟 Bonus possibles

Si tu termines tôt ou si tu veux pousser :

- **Bouton Copier** — copie le mot de passe dans le presse-papiers (`Clipboard.SetText(txtResultat.Text);`)
- **Bouton Effacer** — vide le résultat et remet les options par défaut
- **Indicateur de force** — change la couleur de fond du résultat (rouge si court, orange si moyen, vert si long et varié)
- **Au moins un de chaque** — force le mot de passe à contenir au moins un caractère de chaque catégorie cochée (utile pour les sites qui exigent ça)
- **Polissage** — police monospace pour le résultat, alignement des contrôles, marges, raccourci clavier `Entrée` pour générer

---

✅ Critères de réussite

- ✅ L'interface est claire et lisible
- ✅ Les contrôles sont **renommés** avec des préfixes cohérents
- ✅ Le bouton **Générer** fonctionne
- ✅ Les options sont respectées dans le résultat

- ✅ Les cas d'erreur affichent un message utile
- ✅ Le code est **indenté** et lisible
- ✅ Les tests du tableau ci-dessus passent tous

💡 Conseils de méthode

Travaille **dans cet ordre**, pas autrement :

1. **Interface** seulement, sans aucun code
2. **Test du bouton** avec un simple `MessageBox.Show`
3. **Lecture** des valeurs (avec un MessageBox temporaire pour vérifier)
4. **Logique** de génération
5. **Affichage** du résultat
6. **Gestion d'erreur**
7. **Polissage** et bonus

À chaque étape, lance l'application et vérifie. Si ça marche, passe à la suivante. Si ça ne marche pas, ne va pas plus loin — **règle le problème immédiatement**. Empiler des bugs sans les voir, c'est la recette pour ne plus savoir où chercher.

🤔 Défi réflexion

Réponds par écrit à la question suivante en quelques lignes :

> ***Qu'est-ce qui rend un mot de passe sécurisé ?***

Dans ta réponse, explique :

- le rôle de la **longueur** — pourquoi un mot de passe de 6 caractères est-il facile à casser, même mixte ?
- l'intérêt de mélanger les **types de caractères** — qu'est-ce que ça change pour un attaquant ?
- les **limites** d'un mot de passe aléatoire — qu'est-ce qui ne dépend pas du mot de passe lui-même (fuites de données, phishing, gestionnaire de mots de passe…) ?

✍️ À retenir

- Une application **Windows Forms** est composée d'une fenêtre (`Form`), de contrôles posés dessus, et de méthodes C# qui réagissent à des **événements**.
- Le **concepteur visuel** de Rider permet de dessiner l'interface à la souris ; le code derrière se trouve dans `Form1.cs` .
- **Renommer ses contrôles** avec une convention (`btn` , `txt` , `chk` , `lbl` , `num` …) est indispensable pour rester lisible.

- Double-cliquer sur un contrôle crée la méthode liée à son événement principal (`Click` pour un bouton).
- Toujours **valider les entrées** utilisateur avant de les utiliser, avec un message d'erreur clair en cas de problème.
- Avancer **par petits pas validés** est plus rapide que coder en bloc et chercher où ça plante.

Les conventions de nommage en C#

> *Dans cet article, nous allons voir comment nommer correctement les éléments d'un programme C# afin d'écrire un code plus clair, plus professionnel et plus facile à maintenir.*

Pourquoi des conventions de nommage ?

Quand on programme, on ne choisit pas les noms "au hasard". Un bon nom permet de comprendre rapidement :

- ce que représente une variable ;
- à quoi sert une méthode ;
- quel rôle joue une classe ;
- si un élément est privé, public, constant, local, etc.

Des conventions de nommage communes permettent aussi à plusieurs développeurs de lire le même code sans être perturbés par des styles différents.

En C#, ces conventions sont très importantes. Elles rendent le code plus lisible et plus cohérent.

Principe général

Un bon nom doit être :

- clair ;
- précis ;
- facile à lire ;
- lié au rôle réel de l'élément.

On évite donc les noms vagues comme :

```
int x;
string truc;
bool test;
```

On préfère :

```
int age;
string nomJoueur;
bool estVivant;
```

Deux styles à connaître absolument

En C#, il existe deux grandes conventions d'écriture pour les noms.

PascalCase

Chaque mot commence par une majuscule.

Exemples :

```
NomJoueur
PointsDeVie
AfficherPersonnage
```

camelCase

Le premier mot commence par une minuscule, puis chaque mot suivant commence par une majuscule.

Exemples :

```
nomJoueur
pointsDeVie
afficherPersonnage
```

Règle essentielle en C#

En C#, on utilise généralement :

- **PascalCase** pour les classes, méthodes, propriétés ;
- **camelCase** pour les variables locales et les paramètres.

Nommer une classe

Une classe représente souvent une entité, un objet ou un concept. Son nom doit donc être un nom clair, souvent au singulier.

On utilise **PascalCase**.

Exemples corrects :

```
class Personnage
class CompteBancaire
class CapteurTemperature
class Produit
```

Exemples moins bons :

```
class personnage
class perso
class machin
```

Conseil

Le nom d'une classe doit en général répondre à la question :

> *"Quel type d'objet est-ce ?"*

Exemples :

- `Personnage`
- `Voiture`
- `Eleve`
- `Commande`

Nommer un objet ou une variable

Une variable contient une valeur. Son nom doit décrire cette valeur.

On utilise généralement **camelCase**.

Exemples :

```
string nom;
int pointsDeVie;
bool estVivant;
double temperatureActuelle;
```

Mauvais exemples

```
string a;
int valeur1;
bool b;
```

Ces noms ne disent presque rien.

Nommer une méthode

Une méthode représente une action. Son nom doit donc, en général, commencer par un verbe.

On utilise **PascalCase**.

Exemples :

```
AfficherPersonnage()
Attaquer()
RecevoirDegats()
CalculerTotal()
Demarrer()
```

Idée importante

Le nom d'une méthode doit répondre à la question :

> *"Que fait cette méthode ?"*

On évite donc les noms trop flous comme :

```
Faire()
Traiter()
Go()
Action()
```

sauf si le contexte est vraiment évident.

Nommer une propriété

Une propriété représente souvent une caractéristique d'un objet.

On utilise **PascalCase**.

Exemples :

```
public string Nom { get; set; }
public int PointsDeVie { get; set; }
public bool EstVivant { get; set; }
```

Même si, dans d'autres langages, on voit souvent des attributs en minuscule, en C# une propriété publique se nomme en général comme une classe ou une méthode : avec **PascalCase**.

Nommer un champ privé

Un champ privé est une variable déclarée dans une classe, souvent utilisée en interne.

Il existe plusieurs styles selon les équipes, mais une convention très fréquente en C# est :

- **underscore + camelCase**

Exemples :

```
private string _nom;
private int _pointsDeVie;
private bool _estVivant;
```

Cela permet de repérer immédiatement qu'il s'agit d'un champ privé.

Exemple avec propriété :

```
class Personnage
{
    private int _pointsDeVie;

    public int PointsDeVie
    {
        get { return _pointsDeVie; }
        set
        {
            if (value >= 0)
            {
                _pointsDeVie = value;
            }
        }
    }
}
```

Nommer un paramètre

Un paramètre est une donnée reçue par une méthode.

On utilise généralement **camelCase**.

Exemple :

```
public void RecevoirDegats(int degats)
{
    pointsDeVie -= degats;
}
```

Autres exemples :

```
public void ChangerNom(string nouveauNom)
public int Additionner(int nombre1, int nombre2)
```

Nommer une constante

Une constante est une valeur qui ne change pas.

En C#, on utilise généralement **PascalCase** pour les constantes, même si dans d'autres langages on voit parfois du MAJUSCULE_AVEC_UNDERSCORES.

Exemple :

```
const int PointsDeVieMaximum = 100;
const double Tva = 0.21;
```

Les noms booléens

Les variables booléennes ne contiennent que deux valeurs : `true` ou `false`.

Leur nom doit donc exprimer clairement un état ou une question.

Exemples :

```
bool estVivant;
bool estConnecte;
bool aGagne;
bool peutAttaquer;
```

On reconnaît souvent un booléen grâce à des débuts comme :

- `est ...`
- `a ...`
- `peut ...`
- `doit ...`

Cela rend les conditions beaucoup plus lisibles :

```
if (estVivant)
{
    Console.WriteLine("Le personnage peut continuer.");
}
```

Éviter les abréviations inutiles

Les abréviations rendent souvent le code moins clair.

Exemple moins bon :

```
int nbPv;
string nomPers;
```

Exemple plus clair :

```
int nombreDePointsDeVie;
string nomPersonnage;
```

Cela dit, certaines abréviations très connues peuvent rester acceptables si elles sont comprises par tous, par exemple :

- `id`
- `url`
- `tv`
- `xml`

Mais il faut rester prudent.

Éviter les noms trop longs

Un nom doit être précis, mais pas interminable.

Exemple excessif :

```
int nombreTotalDePointsDeVieRestantsDuPersonnagePrincipal;
```

Exemple plus raisonnable :

```
int pointsDeVieRestants;
```

Le but est de trouver un équilibre entre précision et lisibilité.

Une idée par nom

Un nom doit représenter une seule chose claire.

Exemple confus :

```
string infos;
```

Que contient `infos` ? Un nom ? Un texte complet ? Une liste ? Une description ?

On préfère :

```
string description;
string nomComplet;
string messageErreur;
```

Respecter la langue choisie

Il faut éviter de mélanger plusieurs langues dans le même code.

Exemple à éviter :

```
string playerName;
int pointsDeVie;
bool isAlive;
```

Mieux vaut choisir une convention cohérente :

Tout en français

```
string nomJoueur;
int pointsDeVie;
bool estVivant;
```

Ou tout en anglais

```
string playerName;
int hitPoints;
bool isAlive;
```

Dans un contexte scolaire, surtout au début, tout en français peut être très utile pour comprendre les concepts. Dans un contexte professionnel, l'anglais est très fréquent.

Exemples complets

Exemple peu lisible

```
class perso
{
    public string n;
    public int pv;

    public void atk()
    {
        Console.WriteLine(n);
    }
}
```

Exemple mieux nommé

```
class Personnage
{
    public string Nom;
    public int PointsDeVie;

    public void AfficherNom()
    {
        Console.WriteLine(Nom);
    }
}
```

▸ Exemple avec propriété et champ privé

```
class Personnage
{
    private int _pointsDeVie;

    public string Nom { get; set; }

    public int PointsDeVie
    {
        get { return _pointsDeVie; }
        set
        {
            if (value >= 0)
            {
                _pointsDeVie = value;
            }
        }
    }

    public void RecevoirDegats(int degats)
    {
        PointsDeVie -= degats;
    }
}
```

Tableau récapitulatif

ÉLÉMENT	CONVENTION FRÉQUENTE	EXEMPLE
Classe	PascalCase	`Personnage`
Méthode	PascalCase	`RecevoirDegats()`
Propriété	PascalCase	`PointsDeVie`
Variable locale	camelCase	`pointsDeVie`
Paramètre	camelCase	`degats`
Champ privé	`_camelCase`	`_pointsDeVie`
Constante	PascalCase	`PointsDeVieMaximum`

Ce qu'il faut retenir

Les conventions de nommage ne sont pas du "décor". Elles servent à écrire un code :

- plus lisible ;
- plus logique ;
- plus professionnel ;
- plus facile à corriger et à faire évoluer.

En C#, retiens surtout ceci :

- **Classes, méthodes, propriétés : PascalCase**
- **Variables locales et paramètres : camelCase**
- **Champs privés : souvent** `_camelCase`
- **Noms clairs, précis, cohérents**

Petit conseil de méthode

Quand tu dois choisir un nom, pose-toi simplement une de ces questions :

- "Qu'est-ce que cette donnée représente ?"
- "Que fait cette méthode ?"
- "Quel type d'objet est-ce ?"
- "Est-ce qu'un autre élève comprendrait ce nom sans explication ?"

Si la réponse est oui, le nom est probablement bon.