# 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()`.

```cs
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 : {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();
```

```
57          }
58      }
```

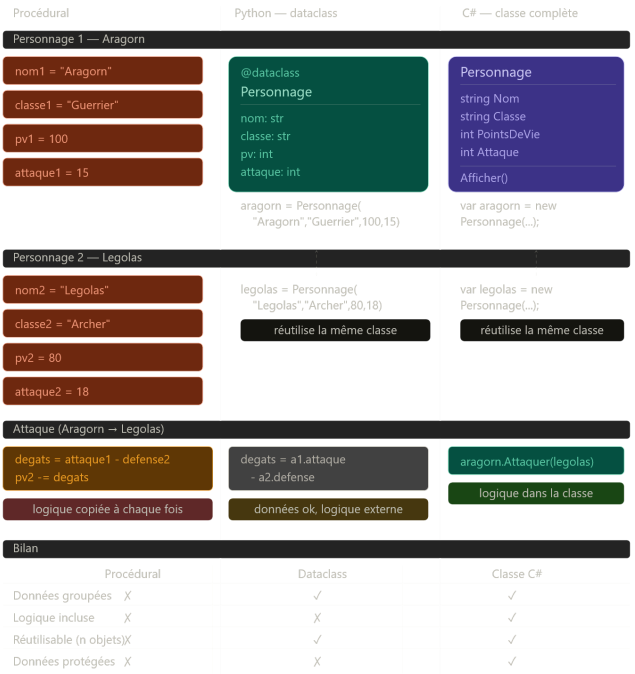
**Résultat :**

```
=== 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*