Chapitre 2 – Encapsulation

Rends-toi compte : avec le code du chapitre 1, une seule ligne suffit pour mettre ton personnage dans un état complètement absurde. hero.Pv = -9999 — et personne ne proteste. C'est le signe qu'un objet mal conçu ne se protège pas lui-même. L'encapsulation consiste à donner à chaque objet la responsabilité de ses propres données : il décide ce qu'on peut lui faire, et il refuse ce qui n'a pas de sens.

Objectifs du chapitre

À la fin de ce chapitre, tu seras capable de :

- Expliquer pourquoi il faut **protéger les données** d'un objet
- Utiliser les mots-clés `private` et `public` correctement
- Écrire des **propriétés C#** avec `get` et `set`
- Ajouter de la **validation** dans un `set`
- Créer des **propriétés calculées** (sans `set`)
- Encapsuler une liste dans une classe `Inventaire`

5 notions-clés

#	Notion	En une phrase
1	`public`	Le membre est accessible depuis n'importe où
2	`private`	L'attribut n'est accessible que depuis l'intérieur de la classe
3	**Propriété**	Syntaxe C# qui remplace les getters/setters tout en gardant le contrôle
4	**Validation**	Code placé dans le `set` pour refuser une valeur incohérente
5	**Propriété calculée**	Propriété en lecture seule dont la valeur est *calculée* à partir d'autres attributs

1. Le problème : rien ne protège les données

Au chapitre 1, on a créé une classe `Personnage` avec des attributs publics. Voici ce que ça permet de faire :

```
Personnage hero = new Personnage();
hero.Nom = "Aragorn";
hero.Pv  = -9999;   // Personne ne proteste. Le compilateur accepte.
hero.Niveau = -42;  // Idem.
```

Le programme compile et tourne sans la moindre erreur. Mais un personnage avec -9999 points de vie, c'est absurde. **Notre objet est dans un état incohérent, et on ne le sait même pas.**

C'est exactement le problème que l'encapsulation résout : **contrôler ce qui entre dans un objet.**

Solution classique : les getters et setters

Dans d'autres langages (Java, PHP…), on résout ce problème en rendant les attributs privés et en créant des méthodes pour y accéder :

```
private int pv;

public int GetPv()
{
    return pv;
}

public void SetPv(int valeur)
{
    if (valeur < 0) valeur = 0;   // on refuse les valeurs négatives
    pv = valeur;
}
```

Ça fonctionne. Mais si la classe a 6 attributs, on se retrouve avec 12 méthodes supplémentaires. C'est lourd à écrire et à lire.

C# propose une syntaxe bien plus élégante pour ça : la *propriété*.

`private` et `public` : la visibilité

Avant de voir les propriétés, il faut comprendre les deux mots-clés de visibilité.

Mot-clé	Accessible depuis	Usage typique
`public`	N'importe où	Méthodes, propriétés
`private`	La classe elle-même uniquement	Attributs internes

La règle générale en POO :

Les attributs sont `private`. Les propriétés et méthodes sont `public`.

On appelle ça le principe d'**encapsulation** : les données sont cachées à l'intérieur de l'objet, et on n'y accède que par une interface contrôlée.

Les propriétés C#

Une **propriété** est une syntaxe C# qui se comporte comme un attribut public pour celui qui l'utilise, mais qui cache un attribut privé à l'intérieur.

Forme complète

```
class Personnage
{
    // Attribut privé (le vrai stockage)
```

```
    private int pv;

    // Propriété publique (l'accès contrôlé)
    public int Pv
    {
        get
        {
            return pv;
        }
        set
        {
            if (value < 0) value = 0;    // validation
            pv = value;
        }
    }
}
```

- Le **`get`** est exécuté quand on *lit* la propriété : `Console.WriteLine(hero.Pv)`
- Le **`set`** est exécuté quand on *écrit* la propriété : `hero.Pv = -9999`
- Le mot-clé **`value`** représente la valeur qu'on essaie d'assigner

Maintenant, si on essaie `hero.Pv = -9999`, la propriété bloque silencieusement et stocke `0` à la place.

Utilisation côté `Program.cs`

```
Personnage hero = new Personnage();
hero.Pv = 100;     // appelle le set → stocke 100
hero.Pv = -9999;   // appelle le set → stocke 0 (validation active)

Console.WriteLine(hero.Pv);   // appelle le get → affiche 0
```

Du point de vue de `Program.cs`, rien ne change. On utilise toujours le `.` comme avec un attribut normal. **La protection est invisible de l'extérieur.**

La classe `Personnage` entièrement encapsulée

Voici la classe complète avec tous ses attributs protégés :

```
class Personnage
{
    // Attributs privés
    private string nom;
    private int pv;
    private int pvMax;
    private int niveau;

    // Propriété : Nom
    public string Nom
    {
        get { return nom; }
```

```
13          set
14          {
15              if (value == "" || value == null) value = "Inconnu";
16              nom = value;
17          }
18      }
19
20      // Propriété : Pv
21      public int Pv
22      {
23          get { return pv; }
24          set
25          {
26              if (value < 0)     value = 0;
27              if (value > pvMax) value = pvMax;
28              pv = value;
29          }
30      }
31
32      // Propriété : PvMax
33      public int PvMax
34      {
35          get { return pvMax; }
36          set
37          {
38              if (value < 1) value = 1;
39              pvMax = value;
40          }
41      }
42
43      // Propriété : Niveau
44      public int Niveau
45      {
46          get { return niveau; }
47          set
48          {
49              if (value < 1) value = 1;
50              niveau = value;
51          }
52      }
53
54      // Propriété calculée : pas d'attribut privé, pas de set
55      public bool EstVivant
56      {
57          get { return pv > 0; }
58      }
59
60      // Méthodes
61      public void AfficherInfos()
62      {
63          string etat = EstVivant ? "En vie" : "Mort";
64          Console.WriteLine($"{Nom} | Niv.{Niveau} | PV : {Pv}/{PvMax} | {etat}");
65      }
66
67      public void RecevoirDegats(int degats)
68      {
69          Pv -= degats;   // passe par la propriété → validation automatique
```

```
        Console.WriteLine($"{Nom} reçoit {degats} dégâts. PV restants : {Pv}");
    }
}
```

Test dans `Program.cs`

```
Personnage hero = new Personnage();
hero.Nom    = "Aragorn";
hero.PvMax  = 100;
hero.Pv     = 100;
hero.Niveau = 5;

hero.AfficherInfos();
// Aragorn | Niv.5 | PV : 100/100 | En vie

hero.RecevoirDegats(30);
// Aragorn reçoit 30 dégâts. PV restants : 70

hero.Pv = -9999;
hero.AfficherInfos();
// Aragorn | Niv.5 | PV : 0/100 | Mort

Console.WriteLine(hero.EstVivant);   // False
```

Propriété calculée en détail

Une **propriété calculée** n'a pas d'attribut privé correspondant. Sa valeur est calculée à la volée à partir d'autres données.

```
public bool EstVivant
{
    get { return pv > 0; }
}
```

Elle n'a **pas de `set`** : personne ne peut écrire `hero.EstVivant = true`. La seule façon de changer `EstVivant`, c'est de modifier `Pv`.

Autres exemples de propriétés calculées 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) + "]";
    }
}
```

Propriétés auto-implémentées (raccourci)

Quand une propriété n'a **pas besoin de validation**, C# propose une syntaxe raccourcie — C# gère l'attribut privé tout seul, en coulisses :

```
// Syntaxe raccourcie (pas de validation possible)
public string Classe { get; set; }
public int Identifiant { get; private set; }   // lecture publique, écriture privée
```

Règle pratique : utilise la forme raccourcie pour les données simples sans contrainte. Utilise la forme complète dès que tu as besoin de valider ou de calculer.

Livrable — La classe `Inventaire`

La classe `Inventaire` contient une liste d'objets appartenant à un personnage. La liste est **privée** : on ne peut pas la modifier directement de l'extérieur. On passe obligatoirement par les méthodes prévues.

```
class Inventaire
{
    // La liste est privée : personne ne peut y accéder directement
    private List<string> objets;

    // Propriété calculée : nombre d'objets
    public int Taille
    {
        get { return objets.Count; }
    }

    // Propriété calculée : est-ce que l'inventaire est plein ?
    public bool EstPlein
    {
        get { return objets.Count >= 10; }
    }

    // Constructeur : initialise la liste vide
    public Inventaire()
    {
        objets = new List<string>();
    }

    // Ajouter un objet (avec vérification)
    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.");
    }

    // Retirer un objet (avec vérification)
    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.");
    }

    // Afficher le contenu
    public void Afficher()
    {
        if (Taille == 0)
        {
            Console.WriteLine("L'inventaire est vide.");
            return;
        }
        Console.WriteLine($"Inventaire ({Taille}/10) :");
        foreach (string o in objets)
        {
            Console.WriteLine("  - " + o);
        }
    }
}
```

Test dans `Program.cs`

```
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
```

```
15
16  Console.WriteLine("Objets restants : " + sac.Taille);
```

Schéma récapitulatif

```
AVANT (chapitre 1)              APRÈS (chapitre 2)
──────────────────              ──────────────────

public string Nom;      →       private string nom;
public int Pv;          →       private int pv;
                                public string Nom  { get; set { validation } }
                                public int Pv      { get; set { validation } }
                                public bool EstVivant { get { calcul } }

hero.Pv = -9999;  ✓             hero.Pv = -9999;  → stocke 0 automatiquement
```

Ce qu'on verra au chapitre suivant

Nos objets peuvent encore se retrouver dans un état incohérent **à la création** : un personnage sans nom, avec `PvMax = 0`… Le constructeur résoudra ce problème en garantissant qu'un objet naît toujours dans un état valide.