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**).

5TTR 6TTR Découverte

*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.GetPvMax()
// 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`.