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.

5TTR 6TTR Découverte

*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 :

```
1  public int Soigner(int montant)
2  {
3      int avant = pv;
4      Pv += montant;
5      int reellement = Pv - avant;
6      return reellement;
7  }
```

Elle se découpe en **cinq parties** :

© 2025 Mr. Lambrechts

```
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.

© 2025 Mr. Lambrechts

```
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);
```

© 2025 Mr. Lambrechts

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` !).

© 2025 Mr. Lambrechts

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;
```

© 2025 Mr. Lambrechts

```
5
6   Personnage legolas = new Personnage();
7   legolas.Nom = "Legolas";  legolas.Classe = "Archer";
8   legolas.PvMax = 80;   legolas.Pv = 80;
9   legolas.Attaque = 18; legolas.Defense = 7; legolas.Niveau = 5;
10
11  Console.WriteLine(aragorn);
12  Console.WriteLine(legolas);
13
14  aragorn.Attaquer(legolas);
15  legolas.Attaquer(aragorn);
16
17  int soin = aragorn.Soigner(15);
18  Console.WriteLine($"{aragorn.Nom} récupère {soin} PV.");
19
20  Console.WriteLine(aragorn);
21  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

© 2025 Mr. Lambrechts

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.

© 2025 Mr. Lambrechts

© 2025 Mr. Lambrechts
https://v2.ttrinfo.be
10 | 10