C# — Todo List

Solution guidée pas à pas

5TTR / 6TTR • v2.ttrinfo.be

Avant de commencer

Ce document te guide pas à pas dans la construction d'une application console de gestion de tâches (Todo List) en C#. Chaque étape est expliquée en détail : pourquoi ce choix, comment ça fonctionne, et quelles alternatives existent.

Lis bien chaque étape avant de coder. Teste après chaque étape avant de passer à la suivante.

Ce que tu vas apprendre

Structurer un programme avec une boucle principale (menu)

Utiliser une List<string> pour stocker des données dynamiques

Lire et valider les saisies de l'utilisateur

Parcourir et modifier une liste avec for et les méthodes .Add() / .RemoveAt()

Gérer les cas d'erreur (liste vide, numéro invalide)

Le programme final permettra à l'utilisateur de :

- Voir ses tâches numérotées
- Ajouter une nouvelle tâche
- Supprimer une tâche terminée
- Quitter proprement

Étape 1 — Afficher le menu en boucle

Objectif de cette étape

Afficher le menu à l'écran, lire le choix de l'utilisateur, et recommencer indéfiniment — jusqu'à ce qu'il choisisse de quitter.

La structure : une boucle infinie + break

On a besoin d'une boucle qui tourne en permanence. La solution classique en C# est :

```
while (true)
{
    // code du menu

    if (choix == "0")
    {
        break;
    }
}
```

🕮 Notion : while(true) et break

while(true) crée une boucle dont la condition est toujours vraie : elle ne s'arrête jamais d'elle-même.

break interrompt la boucle immédiatement, quelle que soit la condition.

C'est le pattern standard pour un menu interactif : on boucle jusqu'à ce que l'utilisateur choisisse de quitter.

Alternative possible : `bool continuer = true; while (continuer) { ... continuer = false; }` — plus explicite, mais plus verbeux. Les deux approches sont valides.

Afficher le menu et lire le choix

À l'intérieur de la boucle, on affiche les options et on lit ce que l'utilisateur tape :

```
while (true)
{
    Console.WriteLine("╔═══════════════════════════╗");
    Console.WriteLine("║       MA TODO LIST        ║");
    Console.WriteLine("╚═══════════════════════════╝");
```

```
    Console.WriteLine("\n 1. Voir les tâches");
    Console.WriteLine(" 2. Ajouter une tâche");
    Console.WriteLine(" 3. Tâche terminée");
    Console.WriteLine(" 0. Quitter");

    Console.Write("\nVotre choix : ");
    string choix = Console.ReadLine();

    if (choix == "0")
    {
        Console.WriteLine("À bientôt !");
        break;
    }
}
```

Pourquoi string et pas int pour le choix ?

On lit le choix comme une string (texte) et on le compare avec == "0", "1", etc.

Si on utilisait int.Parse() directement, le programme planterait si l'utilisateur tape une lettre.

En gardant le choix en string, on peut le comparer sans risque de crash. C'est une approche défensive.

Test à ce stade

Lance le programme. Tu dois voir le menu s'afficher en boucle. Tape 0 pour quitter.

▶ **Résultat dans la console**

```
╔═══════════════════════════╗
║       MA TODO LIST        ║
╚═══════════════════════════╝

 1. Voir les tâches
 2. Ajouter une tâche
 3. Tâche terminée
 0. Quitter

Votre choix : 0
À bientôt !
```

Étape 2 — Aiguiller selon le choix — le switch

Objectif de cette étape

Détecter ce que l'utilisateur a choisi et exécuter le bon bloc de code. On remplace le simple if par un switch.

🕮 Notion : switch

switch permet de comparer une valeur à plusieurs cas possibles, de façon plus lisible qu'une longue chaîne de if / else if.

Chaque case se termine par break (pour sortir du switch). Le case default s'exécute si aucun autre ne correspond.

Syntaxe : `switch (variable) { case "1": ... break; case "2": ... break; default: ... break; }`

```
switch (choix)
{
    case "1":
        Console.WriteLine("→ Voir les tâches");
        break;

    case "2":
        Console.WriteLine("→ Ajouter une tâche");
        break;

    case "3":
        Console.WriteLine("→ Tâche terminée");
        break;

    case "0":
        Console.WriteLine("À bientôt !");
        break;

    default:
        Console.WriteLine("⚠  Choix invalide. Réessayez.");
        break;
}
```

⚠ Attention : le break dans le switch

Le break ici sort du switch, pas de la boucle while.

Pour quitter la boucle, il faudra ajouter un break supplémentaire après le switch quand choix == "0".

Solution propre : gérer le case "0" avant le switch, comme à l'étape 1.

Restructuration : combiner le switch et la sortie

On déplace la gestion du 0 avant le switch pour que le break du while soit bien placé :

```
// Lire le choix
    Console.Write("\nVotre choix : ");
    string choix = Console.ReadLine();

    if (choix == "0")
    {
        Console.WriteLine("À bientôt !");
        break;   // sort du while
    }

    switch (choix)
    {
        case "1": ... break;
        case "2": ... break;
        case "3": ... break;
        default:
            Console.WriteLine("⚠  Choix invalide.");
            break;
    }
```

Test à ce stade

Teste chaque option : chaque case doit afficher son message de test. Le default doit s'afficher si tu tapes n'importe quoi.

Étape 3 — Déclarer la liste et ajouter une tâche

Objectif de cette étape

Créer la structure de données (une List<string>) et implémenter l'option 2 : ajouter une tâche.

🕮 Notion : List<string>

Un tableau classique (string[]) a une taille fixe définie à la création. On ne peut pas y ajouter ou retirer des éléments facilement.

List<string> est un tableau dynamique : sa taille s'adapte automatiquement. On peut y ajouter avec .Add() et retirer avec .RemoveAt().

Le <string> entre chevrons indique que cette liste ne contient que des strings. On parle de type générique.

Déclaration : `List<string> taches = new List<string>();` — crée une liste vide.

Propriété utile : `taches.Count` — donne le nombre d'éléments dans la liste.

Déclarer la liste

La liste doit être déclarée avant la boucle while, pour exister tout au long du programme :

```
// Déclaration de la liste — avant le while
List<string> taches = new List<string>();

while (true)
{
    // ... menu ...
}
```

⚠ Portée des variables

Si tu déclares la liste à l'intérieur du while, elle est recréée (vide) à chaque tour de boucle !

Déclare toujours les données qui doivent persister EN DEHORS de la boucle.

Implémenter l'option 2 — Ajouter une tâche

Dans le case "2" du switch, on demande le texte de la tâche et on l'ajoute à la liste :

```
    case "2":
        Console.Write("Nouvelle tâche : ");
```

```
            string nouvelle = Console.ReadLine();
            taches.Add(nouvelle);
            Console.WriteLine("☑ Tâche ajoutée !");
            break;
```

Test à ce stade

Ajoute quelques tâches et vérifie que le programme ne plante pas. La liste ne s'affiche pas encore — c'est normal, c'est l'étape suivante.

Étape 4 — Afficher la liste des tâches

Objectif de cette étape

Parcourir la liste et afficher chaque tâche numérotée. Gérer le cas où la liste est vide.

> **Notion : parcourir une List avec for**
>
> Les éléments d'une List sont indexés à partir de 0 : taches[0] est le premier, taches[1] le deuxième, etc.
>
> On utilise taches.Count pour connaître le nombre d'éléments et savoir quand s'arrêter.
>
> **Syntaxe classique :** `for (int i = 0; i < taches.Count; i++) { ... taches[i] ... }`
>
> Afficher le numéro lisible pour l'utilisateur : i+1 (car l'utilisateur s'attend à commencer à 1, pas à 0).

```
    case "1":
        Console.WriteLine("--- Mes tâches ---");

        if (taches.Count == 0)
        {
            Console.WriteLine("(aucune tâche pour l'instant)");
        }
        else
        {
            for (int i = 0; i < taches.Count; i++)
            {
                Console.WriteLine($" {i+1}. {taches[i]}");
            }
        }
        break;
```

Test à ce stade

Ajoute 3 tâches puis choisis l'option 1. Tu dois voir :

▶ Résultat dans la console

```
--- Mes tâches ---
 1. Faire les courses
 2. Réviser C#
 3. Promener le chien
```

Teste aussi l'option 1 sans avoir ajouté de tâche :

▶ Résultat dans la console

```
--- Mes tâches ---
(aucune tâche pour l'instant)
```

Notion complémentaire : foreach

C# propose une autre façon de parcourir une liste : la boucle foreach. C'est souvent plus lisible que for quand on n'a pas besoin de l'index.

🕮 Notion : foreach

foreach parcourt automatiquement tous les éléments d'une collection, un par un, sans gérer de compteur.

Syntaxe : `foreach (string tache in taches) { ... }`

Lecture : "pour chaque tache dans taches, fais..."

La variable de boucle (ici tache) prend automatiquement la valeur de chaque élément à chaque tour.

for vs foreach — quand utiliser quoi ?

Utilise for quand...	Utilise foreach quand...
Tu as besoin du numéro (index)	Tu veux juste lire chaque élément
Tu modifies ou supprimes des éléments	Tu parcours pour afficher ou calculer
Tu compares deux positions dans la liste	Le code gagne en lisibilité

Exemple comparé

Afficher la liste avec for (quand on a besoin du numéro) :

```
for (int i = 0; i < taches.Count; i++)
{
    Console.WriteLine($" {i+1}. {taches[i]}");
}
```

La même chose avec foreach (si on n'avait pas besoin du numéro) :

```
foreach (string tache in taches)
{
    Console.WriteLine($" - {tache}");
}
```

▶ Résultat dans la console

```
 - Faire les courses
 - Réviser C#
```

```
  - Promener le chien
```

> ⚠ **Limitation importante du foreach**
>
> Tu ne peux PAS modifier la liste pendant un foreach (ajouter, supprimer un élément).
>
> C# lève une exception si tu essaies : "Collection was modified; enumeration operation may not execute."
>
> Pour toute opération qui modifie la liste, utilise obligatoirement un for.

Dans notre Todo List

On garde le for pour l'affichage numéroté (étape 4) et la suppression (étape 5), car on a besoin de l'index dans les deux cas.

Le foreach serait idéal si on ajoutait, par exemple, une option `"Tout afficher sans numéros"` ou une recherche dans les tâches.

Étape 5 — Supprimer une tâche terminée

Objectif de cette étape

Permettre à l'utilisateur de choisir une tâche par son numéro et la retirer de la liste.

> 🕮 **Notion : RemoveAt()**
>
> taches.RemoveAt(i) retire l'élément à l'index i. Les éléments suivants se décalent automatiquement.
>
> **Attention à la conversion d'index :** l'utilisateur tape 1, 2, 3... mais les index de la liste commencent à 0.
>
> **Conversion :** `index = numéro_saisi - 1`

```
    case "3":
        if (taches.Count == 0)
        {
            Console.WriteLine("⚠  Aucune tâche dans la liste.");
            break;
        }

        // Afficher la liste pour aider l'utilisateur
        Console.WriteLine("--- Mes tâches ---");
        for (int i = 0; i < taches.Count; i++)
            Console.WriteLine($" {i+1}. {taches[i]}");

        Console.Write("Tâche terminée (numéro) : ");
        int numero = int.Parse(Console.ReadLine());

        if (numero < 1 || numero > taches.Count)
        {
            Console.WriteLine("⚠  Numéro invalide.");
        }
        else
        {
            string tacheRetiree = taches[numero - 1];
            taches.RemoveAt(numero - 1);
            Console.WriteLine($"☑ \"{tacheRetiree}\" retirée de la
liste.");
        }
        break;
```

Pourquoi stocker la tâche avant de la supprimer ?

On sauvegarde le texte dans `tacheRetiree` AVANT d'appeler RemoveAt() — parce qu'après la suppression, l'élément n'existe plus dans la liste.

C'est une bonne habitude : récupère ce dont tu as besoin avant de modifier la structure.

Test à ce stade

Ajoute 3 tâches, puis retire la numéro 2. Vérifie que la liste est bien mise à jour :

▶ Résultat dans la console

```
--- Mes tâches ---
 1. Faire les courses
 2. Réviser C#
 3. Promener le chien

Tâche terminée (numéro) : 2
☑ "Réviser C#" retirée de la liste.
```

Étape 6 — Nettoyer la console entre les affichages

Objectif de cette étape

Éviter que la console s'accumule et devienne illisible — effacer l'écran avant chaque affichage du menu.

```
while (true)
{
    Console.Clear();    // efface la console

    // afficher le menu...
}
```

Console.Clear()

Console.Clear() efface tout ce qui est affiché dans la fenêtre de console.

Placé au début du while, il garantit que l'écran est propre avant chaque nouveau cycle du menu.

C'est un détail simple mais qui rend l'application nettement plus agréable à utiliser.

Étape ✓ — Code complet — version finale

Voici le programme complet, assemblé et commenté :

```
// Todo List — version finale

List<string> taches = new List<string>();

while (true)
{
    Console.Clear();

    Console.WriteLine("╔═══════════════════════════╗");
    Console.WriteLine("║       MA TODO LIST        ║");
    Console.WriteLine("╚═══════════════════════════╝");
    Console.WriteLine("\n 1. Voir les tâches");
    Console.WriteLine(" 2. Ajouter une tâche");
    Console.WriteLine(" 3. Tâche terminée");
    Console.WriteLine(" 0. Quitter");
    Console.Write("\nVotre choix : ");
    string choix = Console.ReadLine();

    if (choix == "0")
    {
        Console.WriteLine("À bientôt !");
        break;
    }

    switch (choix)
    {
        case "1":
            Console.WriteLine("--- Mes tâches ---");
            if (taches.Count == 0)
                Console.WriteLine("(aucune tâche pour l'instant)");
            else
                for (int i = 0; i < taches.Count; i++)
                    Console.WriteLine($" {i+1}. {taches[i]}");
            Console.ReadKey();  // pause
            break;

        case "2":
            Console.Write("Nouvelle tâche : ");
            string nouvelle = Console.ReadLine();
            taches.Add(nouvelle);
```

```
            Console.WriteLine("☑ Tâche ajoutée !");
            Console.ReadKey();
            break;

        case "3":
            if (taches.Count == 0)
            {  Console.WriteLine("⚠  Aucune tâche."); break; }
            for (int i = 0; i < taches.Count; i++)
                Console.WriteLine($" {i+1}. {taches[i]}");
            Console.Write("Tâche terminée (numéro) : ");
            int numero = int.Parse(Console.ReadLine());
            if (numero < 1 || numero > taches.Count)
                Console.WriteLine("⚠  Numéro invalide.");
            else {
                string t = taches[numero-1];
                taches.RemoveAt(numero-1);
                Console.WriteLine($"☑ \"{t}\" retirée."); }
            Console.ReadKey();
            break;

        default:
            Console.WriteLine("⚠  Choix invalide.");
            Console.ReadKey();
            break;
    }
}
```

Console.ReadKey() — la pause

Après chaque action, on appelle Console.ReadKey() : le programme attend que l'utilisateur appuie sur une touche.

Sans cette pause, Console.Clear() effacerait immédiatement le résultat avant que l'utilisateur ait pu le lire.

Pour aller plus loin

Tu as terminé le programme de base. Voici quelques pistes pour l'améliorer :

Défi 1 — Empêcher les tâches vides

Si l'utilisateur appuie sur Entrée sans rien taper, la tâche ajoutée est une chaîne vide.

Hint : `string.IsNullOrWhiteSpace(nouvelle)` retourne true si la chaîne est vide ou ne contient que des espaces.

Défi 2 — Numéro saisi sans planter

Actuellement, si l'utilisateur tape une lettre à l'option 3, le programme plante sur int.Parse().

Hint : `int.TryParse(texte, out int valeur)` retourne true si la conversion réussit, sans lever d'exception.

Défi 3 — Sauvegarder dans un fichier

Utilise `File.WriteAllLines("taches.txt", taches)` pour sauvegarder la liste, et `File.ReadAllLines("taches.txt")` pour la recharger au démarrage.

Défi 4 — Modifier une tâche

Ajoute une option 4 qui permet de remplacer le texte d'une tâche existante par un nouveau texte.

Hint : `taches[index] = nouveauTexte;`

Bonus — Refactoring : extraire des fonctions

Le programme fonctionne, mais tout le code est dans la boucle principale. Quand un programme grandit, ça devient vite difficile à lire et à maintenir.

La solution : extraire des blocs de code dans des fonctions réutilisables.

> **📖 Notion : les fonctions (méthodes statiques)**
>
> Une fonction est un bloc de code nommé qu'on peut appeler autant de fois qu'on veut.
>
> En C#, en dehors des classes, on déclare des fonctions avec static void NomDeLaFonction() { ... }
>
> **Si la fonction a besoin de données :** on les passe en paramètres — comme en Python ou PHP.
>
> **Si la fonction doit renvoyer un résultat :** on remplace void par le type de retour et on utilise return.
>
> Pour l'instant on utilise void (pas de retour) — on verra les fonctions avec retour en détail dans un prochain chapitre.

AfficherMenu()

Le code d'affichage du menu n'a besoin d'aucune donnée externe — il affiche toujours la même chose. C'est le candidat idéal pour une fonction sans paramètre :

```
static void AfficherMenu()
{
    Console.WriteLine("╔═══════════════════════════╗");
    Console.WriteLine("║       MA TODO LIST        ║");
    Console.WriteLine("╚═══════════════════════════╝");
    Console.WriteLine("\n 1. Voir les tâches");
    Console.WriteLine(" 2. Ajouter une tâche");
    Console.WriteLine(" 3. Tâche terminée");
    Console.WriteLine(" 0. Quitter");
    Console.Write("\nVotre choix : ");
}
```

Dans la boucle principale, on remplace tout le bloc d'affichage par un simple appel :

```
while (true)
{
    Console.Clear();
    AfficherMenu();   // ← une ligne au lieu de 8
    string choix = Console.ReadLine();
    ...
```

```
}
```

AfficherListeTaches()

L'affichage de la liste est utilisé deux fois : dans l'option 1 et dans l'option 3 (avant de choisir un numéro). C'est exactement le cas où une fonction évite la duplication.

Cette fois, la fonction a besoin de la liste — on la passe en paramètre :

```
static void AfficherListeTaches(List<string> taches)
{
    Console.WriteLine("--- Mes tâches ---");

    if (taches.Count == 0)
    {
        Console.WriteLine("(aucune tâche pour l'instant)");
    }
    else
    {
        for (int i = 0; i < taches.Count; i++)
            Console.WriteLine($" {i+1}. {taches[i]}");
    }
}
```

Dans le switch, les deux options deviennent :

```
    case "1":
        AfficherListeTaches(taches);
        Console.ReadKey();
        break;

    case "3":
        if (taches.Count == 0) { ... break; }
        AfficherListeTaches(taches);   // réutilisée ici aussi
        ...
```

Principe DRY — Don't Repeat Yourself

Si tu copies-colles le même bloc de code à deux endroits, c'est le signal qu'il faut en faire une fonction.

Pourquoi ? Si tu dois corriger un bug ou modifier l'affichage, tu ne le fais qu'à un seul endroit.

C'est l'un des principes fondamentaux de la programmation, quel que soit le langage.

EffacerTache() — ça a du sens ?

On pourrait extraire la logique de suppression dans une fonction. Voici ce que ça donnerait :

```
static void EffacerTache(List<string> taches)
{
    if (taches.Count == 0)
    {
        Console.WriteLine("⚠  Aucune tâche dans la liste.");
        return;   // sort de la fonction immédiatement
    }

    AfficherListeTaches(taches);

    Console.Write("Tâche terminée (numéro) : ");
    int numero = int.Parse(Console.ReadLine());

    if (numero < 1 || numero > taches.Count)
    {
        Console.WriteLine("⚠  Numéro invalide.");
        return;
    }

    string t = taches[numero - 1];
    taches.RemoveAt(numero - 1);
    Console.WriteLine($"☑ \"{t}\" retirée de la liste.");
}
```

return dans une fonction void

Dans une fonction void (sans valeur de retour), return seul permet de sortir prématurément de la fonction.

C'est utile pour gérer les cas d'erreur en tête de fonction — on vérifie les conditions limites, et si quelque chose ne va pas, on sort immédiatement sans exécuter le reste.

C'est plus lisible qu'un grand if / else qui englobe tout le code.

Le case "3" du switch se réduit alors à :

```
    case "3":
        EffacerTache(taches);
        Console.ReadKey();
        break;
```

⚠ List passée par référence

En C#, une List<string> passée en paramètre n'est PAS copiée : la fonction travaille sur la même liste que le programme principal.

Quand EffacerTache() appelle taches.RemoveAt(), la liste du programme principal est bien modifiée.

C'est le comportement attendu ici — et c'est différent des types simples (int, string) qui, eux, sont copiés.

Défis avancés

Défi 5 — Couleurs avec ConsoleColor

C# permet de coloriser les textes de la console avec l'enum ConsoleColor (16 couleurs disponibles).

📖 API à utiliser

`Console.ForegroundColor = ConsoleColor.Green;` → couleur du texte

`Console.BackgroundColor = ConsoleColor.DarkBlue;` → couleur de fond

`Console.ResetColor();` → remet les couleurs par défaut (TOUJOURS appeler après)

Objectif : modifier AfficherMenu() et AfficherListeTaches() pour utiliser des couleurs :

- Titre du menu : fond DarkBlue, texte White
- Options du menu : texte Cyan
- Option 0 (Quitter) : texte DarkGray
- Message de succès ✅ : texte Green
- Message d'erreur ⚠️ : texte Red
- Numéros de tâche : texte Yellow

Exemple de départ :

```
Console.ForegroundColor = ConsoleColor.Green;
Console.WriteLine("☑ Tâche ajoutée !");
Console.ResetColor();
```

⚠ Toujours ResetColor()

Si tu oublies Console.ResetColor(), toutes les lignes suivantes héritent de la dernière couleur définie.

Bonne pratique : appelle ResetColor() à la fin de chaque fonction qui change une couleur.

Défi 6 — Interface avec Spectre.Console

Tu connais déjà la librairie Rich en Python. Spectre.Console en est l'équivalent direct pour C# — même philosophie, API similaire.

📦 Installation

Dans le terminal de Rider (onglet Terminal en bas) :

```
dotnet add package Spectre.Console
```

Puis en haut du fichier :

```
using Spectre.Console;
```

Ce que tu peux utiliser dans la Todo List :

```
// Texte coloré avec balises markup (comme Rich)
AnsiConsole.MarkupLine("[bold green]☑ Tâche ajoutée ![/]");
AnsiConsole.MarkupLine("[red]⚠  Numéro invalide.[/]");

// Tableau formaté automatiquement
var table = new Table();
table.AddColumn("N°");
table.AddColumn("Tâche");
for (int i = 0; i < taches.Count; i++)
    table.AddRow($"{i+1}", taches[i]);
AnsiConsole.Write(table);

// Prompt de sélection interactif (flèches du clavier)
string choix = AnsiConsole.Prompt(
    new SelectionPrompt<string>()
        .Title("[bold]Que veux-tu faire ?[/]")
        .AddChoices("Voir les tâches", "Ajouter", "Quitter"));
```

Comparaison Rich (Python) vs Spectre.Console (C#)

Rich Python : `console.print("[bold green]Texte[/]")`

Spectre C# : `AnsiConsole.MarkupLine("[bold green]Texte[/]")`

La syntaxe des balises markup est quasiment identique — si tu connais Rich, tu te sentiras à l'aise.

Documentation complète : spectreconsole.net

Objectif du défi : remplacer l'affichage de la liste par un tableau Spectre.Console, et le menu texte par un SelectionPrompt interactif navigable au clavier.