Référence rapide C#

Types, LINQ, async/await, collections, POO

Bases

Hello World

```
Console.WriteLine("Hello, World!");  // top-level (C# 10+)
// Classique : class Program { static void Main() { ... } }
```

Compiler et exécuter

```
dotnet new console -n MyApp   # créer un projet
dotnet run                     # compiler et exécuter
dotnet build                   # compiler seulement
```

Variables et constantes

```
int x = 42;
var name = "Alice";           // inférence de type
const double Pi = 3.14159;
readonly int maxRetries = 3;  // défini une fois, dans le ctor
```

Types

Types valeur

`int`	Entier signé 32 bits
`long`	Entier signé 64 bits
`float`	Virgule flottante 32 bits (suffixe **`f`**)
`double`	Virgule flottante 64 bits
`decimal`	Haute précision 128 bits (suffixe **`m`**)
`bool`	**`true`** / **`false`**
`char`	Caractère Unicode 16 bits

Types référence

`string`	Texte UTF-16 immuable
`object`	Type de base pour tous les types
`dynamic`	Contourne la vérification de type à la compilation
`int[]`	Tableau d'entiers
`List<T>`	Liste générique (System.Collections.Generic)

Types nullables et tuples

```
int? age = null;              // type valeur nullable
string? name = null;          // référence nullable (C# 8+)
var point = (X: 1, Y: 2);     // tuple nommé
Console.WriteLine(point.X);
```

Fonctionnalités des chaînes

```
string name = "World";
string msg = $"Hello, {name}!";       // interpolation
string path = @"C:\Users\file.txt";   // verbatim
string raw = """raw "string" here""";  // brut (C# 11+)
```

Flux de contrôle

If / Else

```
if (x > 0) Console.WriteLine("positive");
else if (x == 0) Console.WriteLine("zero");
else Console.WriteLine("negative");
```

Switch et correspondance de motifs

```
string label = x switch {
    > 0 => "positive", 0 => "zero", _ => "negative"
};
if (obj is string s && s.Length > 0) { }  // correspondance de motif
```

Boucles

```
for (int i = 0; i < 10; i++) { }
foreach (var item in collection) { }
while (condition) { }
do { } while (condition);
```

Classes

Définition d'une classe

```
public class Person {
    public string Name { get; set; }
    public int Age { get; init; }       // init-only (C# 9+)
    public Person(string name, int age) { Name = name; Age = age; }
}
```

Records (C# 9+)

```
public record Point(double X, double Y);
var p1 = new Point(1, 2);
var p2 = p1 with { X = 3 };  // copie non destructive
// auto : Equals, GetHashCode, ToString, déconstruction
```

Héritage

```
public abstract class Shape { public abstract double Area(); }
public class Circle(double r) : Shape {
    public override double Area() => Math.PI * r * r;
}
```

Modificateurs d'accès

`public`	Accessible depuis n'importe où
`private`	Même classe uniquement (défaut pour les membres)
`protected`	Même classe et classes dérivées
`internal`	Même assembly uniquement (défaut pour les classes)
`protected internal`	Même assembly ou classes dérivées

Interfaces

Définition d'interface

```
public interface IShape {
    double Area();
    double Perimeter() => 0;  // impl par défaut (C# 8+)
}
public class Rect(double w, double h) : IShape { public double
Area() => w * h; }
```

Interfaces courantes

`IEnumerable<T>`	Support de l'itération (foreach, LINQ)
`IDisposable`	Nettoyage déterministe (instruction **`using`**)
`IComparable<T>`	Ordre naturel pour le tri
`IEquatable<T>`	Comparaison d'égalité de valeur
`ICloneable`	Clonage d'objet

LINQ

Syntaxe de méthode

```
var result = numbers
    .Where(n => n > 3)
    .OrderBy(n => n)
    .Select(n => n * 2)
    .ToList();
```

Syntaxe de requête

```
var result = from n in numbers
             where n > 3
             orderby n
             select n * 2;
```

Méthodes LINQ courantes

`.Where(pred)`	Filtrer les éléments
`.Select(func)`	Projeter/transformer les éléments
`.OrderBy(key)`	Trier en ordre croissant
`.GroupBy(key)`	Grouper les éléments par clé
`.First() / .FirstOrDefault()`	Premier élément (ou valeur par défaut)
`.Any(pred)`	**`true`** si un élément correspond
`.Count()`	Nombre d'éléments
`.Sum() / .Average()`	Agréger des valeurs numériques
`.Distinct()`	Supprimer les doublons
`.SelectMany(func)`	Aplatir des collections imbriquées

Async/Await

Méthode async

```
public async Task<string> FetchAsync(string url) {
    using var client = new HttpClient();
    return await client.GetStringAsync(url);
}
```

Combinateurs de tâches

```
var results = await Task.WhenAll(task1, task2, task3);
var first = await Task.WhenAny(task1, task2);
```

Patterns async

`Task`	Retour async void (sans résultat)
`Task<T>`	Retour async avec résultat de type T
`ValueTask<T>`	Tâche légère pour les chemins synchrones rapides
`await foreach`	Itération async sur **`IAsyncEnumerable<T>`**
`CancellationToken`	Annulation coopérative des opérations async

Collections

Collections courantes

`List<T>`	Tableau dynamique, accès par index rapide
`Dictionary<K,V>`	Table de hachage, O(1) par clé
`HashSet<T>`	Éléments uniques, O(1) de recherche
`Queue<T>`	Collection FIFO
`Stack<T>`	Collection LIFO
`LinkedList<T>`	Liste doublement chaînée
`SortedDictionary<K,V>`	Trié par clé (basé sur arbre)

Utilisation d'un Dictionary

```
var dict = new Dictionary<string, int> {
    ["Alice"] = 90, ["Bob"] = 85
};
dict.TryGetValue("Alice", out int score);
foreach (var (key, val) in dict) { }
```

Collections immuables

```
using System.Collections.Immutable;
var list = ImmutableList.Create(1, 2, 3);
var newList = list.Add(4);  // retourne une nouvelle liste
```

Propriétés

Syntaxe des propriétés

```
public string Name { get; set; }
public int Age { get; private set; }
public string Email { get; init; }        // init-only
public string Display => $"{Name} ({Age})"; // calculée
```

Référence rapide C#

Indexeurs

```
public double this[int row, int col] {
    get => data[row, col];
    set => data[row, col] = value;
}
```

Patterns de propriétés

`{ get; set; }`	Propriété auto en lecture-écriture
`{ get; }`	Lecture seule (défini uniquement dans le constructeur)
`{ get; init; }`	Lecture seule après initialisation (C# 9+)
`{ get; private set; }`	Lecture publique, écriture privée
`=> expression`	Propriété à corps d'expression (calculée)

Exceptions

Try / Catch / Finally

```
try { int result = int.Parse(input); }
catch (FormatException ex) { Console.Error.WriteLine(ex.Message); }
catch (Exception ex) when (ex is not OutOfMemoryException) { }
finally { /* s'exécute toujours */ }
```

Instruction using

```
using var file = File.OpenRead("data.txt");
// file.Dispose() appelé automatiquement à la fin du scope
// équivalent à try/finally avec Dispose()
```

Exceptions courantes

`ArgumentNullException`	Argument null passé à une méthode
`ArgumentOutOfRangeException`	Argument hors de la plage valide
`InvalidOperationException`	Opération invalide pour l'état courant
`NullReferenceException`	Déréférencement d'un objet null
`KeyNotFoundException`	Clé introuvable dans le dictionnaire
`NotImplementedException`	Méthode non encore implémentée

Exception personnalisée

```
public class AppException : Exception {
    public int Code { get; }
    public AppException(string msg, int code)
        : base(msg) { Code = code; }
}
```