Chapitre XI

Mise à niveau C#

Ce chapitre vous fournit une mise à niveau en C#en partant du principe que vous avez pratiqué du Java. Plusieurs concepts essentiels de C# vous sont présentés car il seront susceptibles d'être utilisés dans vos projets. Tous les concepts C# ne sont pas présentés ici. Nous ne parlerons par exemple pas des `types nullables`, des `délégués`, ou des `méthodes d'extension`. D'autres vous seront présentés plus tard, dans des chapitres ultérieurs, en particulier l'API `LINQ`.

1 Le framework DotNet

Le framework DotNet (`.Net`) a été conçu par Microsoft. Il s'agit d'une infrastructure multiplateformes gratuite et open source. Elle contient notamment plusieurs langages de programmation et API ... tous interopérables (capables de dialoguer entre eux). Parmi les langages que nous allons considérer, il y a :

- C# (*C Sharp*), un langage de programmation impérative multiparadigmes, essentiellement objet.
- F# (*F Sharp*), un langage de programmation strictement fonctionnelle.

Il y en a d'autres comme ASP (), Visual Basic, ...

L'intérêt de DotNet est d'une part de constituer un framework intégrant les technologies actuelles (Web, cloud, IA, développement de jeux, applications mobiles ...), d'autre part son interopérabilité entre langages : c'est simple, on peut directement écrire du code F# dans du code C# ! (ex: interop F# dans C#).

2 Création d'un projet C#

Pour développer nos programmes DotNet, nous utiliserons une interface de développement intégrée (IDE), soit *Visual Studio* sous Windows exclusivement (attention, pas *Visual Studio Code*), soit plutôt *IntelliJ Rider* sous toute plateforme.

Néanmoins on peut tout à fait créer un nouveau projet DotNet en ligne de commande. Ce sera même recommandé dans les projets.

Dans un terminal, on peut connaître l'ensemble des types de projets que l'on peut créer, étant donné une installation (donc les API installées ...) à l'aide de la commande `dotnet new list` comme le montre la figure XI.1.

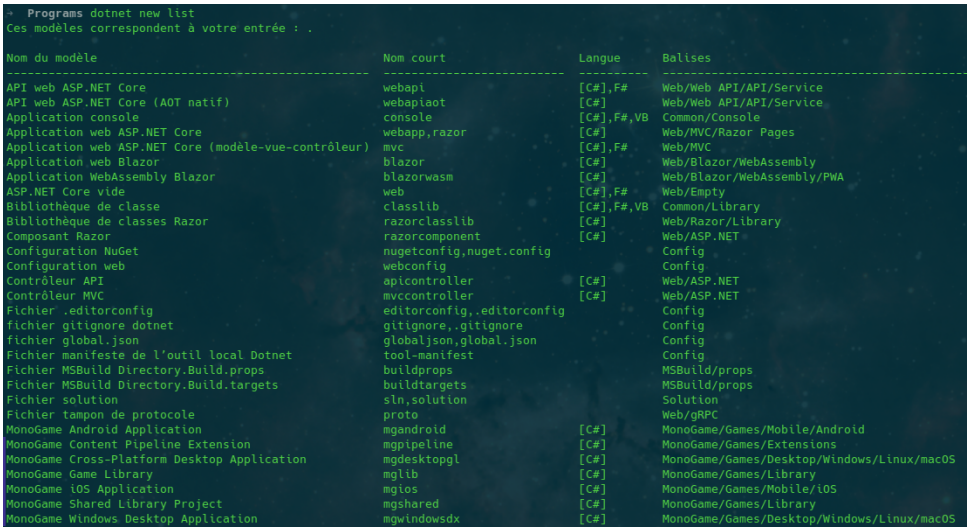

Figure XI.1: Liste de projets dotnet disponibles.

Pour créer un projet propremet dit, on utilise la commande `dotnet new TypeDeProjet -o NomDuProjet` comme l'illustrent les figures XI.2 et XI.3.

Figure XI.2: Création d'un projet DotNet `console`.

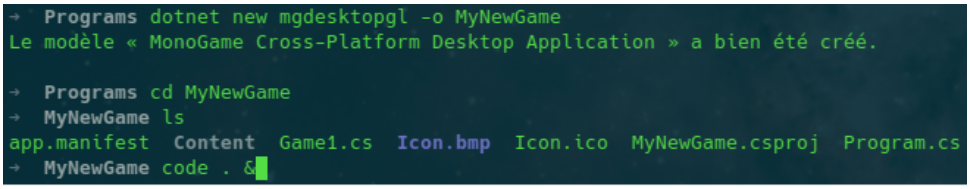

Figure XI.3: Création d'un projet DotNet `console`.

NB: Un projet C# est "habillé" de nombreux fichiers annexes, même un projet de type console. Nous resterons concentrés sur les fichiers d'extension `cs` qui contiennent le code C#.

3 C# - vue d'ensemble

Pour qui a pratiqué Java et, mieux, C++, C# aura des airs de famille. On y retrouve des classes semblables muni de spécificateurs de portée identiques, des types semblables, un même système d'instanciation, une API avec des conteneurs assez proche. C# est néanmoins plus proche de C++ que de Java : le système de généricité est proche bien que moins développé que C++, il est toujours possible de manipuler la mémoire avec des pointeurs, ... mais ... mais l'accent en C# est mis justement sur l'idée d'éviter de descendre dans la programmation bas niveau et, en cela, programmer en C# s'apparente à la manière dont on programme en Java ... en mieux !

Pourquoi mieux ? Parce que Java souffre de nombreux problèmes avec en premier, une retrocompatibilité discutable, mais aussi des incohérences dans la gestion de types, par exemple les conversions non explicites entre types primitifs (comme `double`) et types-objets (resp. `Double`), le fait que ces types-objets ne soient en fait pas instanciés dans le tas (et donc non référençables), mais aussi une généricité extrêmement limitée. Inversement, C# corrige tout cela avec un système de types très élaboré. De plus C# apporte de nombreux ajouts comme les `Propriétés`, une écriture possiblement plus fonctionnelle, la possiblité de disposer de variables immuables, ...

Dans la section qui suit, nous allons découvrir plusieurs aspects du langage C#. Le but de ce cours est que vous soyez rapidement opérationnels en C# en partant du principe que vous connaissez Java. Nous ne verrons donc pas à nouveau comment définir une clase abstraite, une interface, écrire une fonction ..., sans parler des boucles for, while, des switches ... Tout cela, c'est la même chose et vous êtes sensés savoir le faire.

Pour aller plus loin, veuillez vous référer à l'excellente documentation fournie par Microsoft :

- Documentation C#
- Apprentissage C# pour les développeurs Java

4 Ecrire en C#

Dans cette partie, nous allons donc voir un certain nombre de concepts apportés par C# (que l'on retrouve dans certains langages). Nous ne les verrons pas tous évidemment !

4.a Structure d'un programme

Un programme C# peut être constitué d'un seul ou de plusieurs fichiers (extension `cs`). Chaque fichier contient une ou plusieurs classes qui n'ont donc pas nécessairement le même nom que le fichier comme c'est le cas en Java. Le programme contient aussi une classe particulière nommée `Program` qui contient une méthode statique `static void Main(string[] args){ /*code*/ }` ; c'est le point d'entrée du programme comme en Java.

L'ensemble des classes, énumérations, variables de portée globale ..., sont encapsulée dans un `namespace` (comme en C++). Il s'agit de l'équivalent des `package`s en Java. Un `namespace` peut lui-même faire partie d'un plus large `namespace` et ainsi de suite.

Considérons un exemple simple de programme C# et décorticons le. Ceprogramme est constitué de 2 fichiers : un fichier `Distances.cs` qui définit la classe `Distance` et un fichier `Program.cs` qui contient la classe `Program` qui donne son point d'entrée au programme avec la fonction statique `Main`. Remarquez que, contrairement à C++, on ne fait pas d'inclusions de fichiers. On se contente comme en Java d'utiliser des namespaces (importation de packages en Java).

```
1  /* Fichier Distances.cs */
2
3  namespace DistanceNS;
4
5  public sealed class Distance(double value, Distance.Unite unite) {
6      public enum Unite { am, fm, pm, A, nm, mu, mm, cm, inch, dm,
7                          ft, yd, m, km, mile, NM, ua, al, pc, mpc}
8
9      private static readonly double[] UnitValues =
10     { 1E-18, // attomètre
11       1E-15, // femtomètre
12       1E-12, // picomètre
13       1E-10, // Angstrom
14       1E-9, // nanomètre
15       1E-6, // micromètre
16       1E-3, // millimètre
17       0.01, // centimètre
18       0.0254, // inch (pouce)
19       0.1, // décimètre
20       0.3048, // foot (pied)
```

```
      0.9144, // yard
      1.0, // mètre
      1000, // kilomètre
      1609.344, // mille terrestre
      1852, // mille nautique
      149597870700, // unité astronomique
      9460730472580800, // année lumière
      648000 / Double.Pi * 149597870700, // parsec
      648000 / Double.Pi * 149597870700 * 1E6 // mégaparsec
      };
    public double _value { init; get; } = value;
    public Unite _unite { init; get; } = unite;

    private double GetMeters(Unite unite) {
        return UnitValues[(int)unite];
    }

    public Distance ConvertTo(Unite unite) {
        double v = _value * GetMeters(_unite) / GetMeters(unite);
        Distance d = new Distance(v,unite);
        return d;
    }

    public override String ToString() {
        return _value + " " + _unite;
    }
}
```

```
/* Fichier Program.cs */
using DistanceNS;

internal class Program {
    public static void Main(string[] args) {
        Distance dist_1, dist_2;
        dist_1 = new Distance(4.244, Distance.Unite.al); // distance de Proxima
             Centauri
        Console.WriteLine(dist_1); // affichage
        dist_2 = dist_1.ConvertTo(Distance.Unite.pc); // conversion en parsecs
        Console.WriteLine(dist_2); // affichage
        Console.WriteLine(dist_1.ConvertTo(Distance.Unite.km)); // conversion
            en km et affichage
        Console.WriteLine(dist_1.ConvertTo(Distance.Unite.ua)); // conversion
            en unités astronomiques et affichage
    }
}
```

La classe `Distance` permet de définir une distance (ou une longueur) avec son unité. Les unités et leur valeur en mètres sont prédéfinies dans respectivement une énumération et un tableau de valeurs constantes. La classe possède 2 méthodes publiques, une pour la conversion `Distance ConvertTo(Unite unite)` et l'autre pour l'affichage `String ToString()` qui, comme en Java est une surcharge de la méthode `ToString` de base. Elle contient en plus une méthode privée qui permet d'obtenir la valeur en mètres d'une unité donnée. Jusqu'ici, tout cela ressemble fort à du Java.

On voit cependant des différences. Dans cet exemple, il n'y a pas de constructeur et la classe est définie bizarrement, ... comme un constructeur. Il s'agit d'une forme d'écriture compacte permise en C# qui se nomme *constructeur primaire / primary constructor*. Les valeurs passées en paramètres (ici `value` et `unite`) sont affectées à 2 nouvelles structures que vous ne connaissez pas si vous venez de Java ou de C++ : *les propriétés*. Nous décrirons les propriétés dans ce cours, mais remarquez tout de suite qu'elles contiennent 2 groupes de méthodes définies par défaut, ici `init` et `get`. Remarquez comment ces propriétés reçoivent les valeurs passées en paramètres lors de la construction.

Concernant le programme appelant lui-même, pas de grandes surprises : la déclaration et l'instanciation des objets se fait comme en Java/C++. La seule vraie différence concerne la méthode d'affichage qui se nomme `Console.WriteLine`.

La sortie de ce programme (qui convertit la distance de 4.244 années lumières, distance de Proxima du Centaure au système solaire en différentes unités : parsecs, kms, unités astronomiques) est donnée ci-dessous.

```
4,244 al
1,3012163152258764 pc
40151340125632,914 km
268395,1311456261 ua
```

4.b Types

Au lien suivant, vous trouverez tous les détails sur le système de types C#.

En C#, tout est typé ! C'est facile, vous retrouverez les même types primitifs qu'en Java ou C++ (int, double, float, bool, string ...). Il s'agit là de typage *explicite*, c'est-à-dire qu'on exprime dans le code le type que l'on attend, par exemple:

```
string s = "hello world!";
```

Types anonymes (var)

En C#, comme en C++, on peut cependant faire du typage *implicite* avec des `types anonymes`, c'est-à-dire que l'on n'indique pas au préalable le type de la variable ; celui-ci est déduit de la valeur que va prendre cette variable. C'est le compilateur qui réalise ce travail que l'on appelle *inférence de type*. En C# on utilise pour cela le mot `var` (`auto` en C++). Exemple :

```
var s = "hello world!"; // ok, s sera de type string
```

Attention, pour faire cela, il faut absolument que le compilateur C# soit capable d'évaluer le type de la variable lors de la compilation ; autrement dit, il doit être capable de disposer des informations nécessaires sur le type de valeur qu'on affecte à la variable.

Types nullables

Un apport important de C# est la possibilité de définir des variables de type nullable. Une valeur de type nullable est en fait une instance de la structure `System.Nullable<T>`. Les valeurs nullables peuvent accepter une valeur supplémentaire par rapport au type de base, à savoir la valeur `null`. L'intéret est de pouvoir affecter une valeur `null` lorsque la valeur est indéfinie. Cela permet de gérer des cas où des champs (d'une base de donnée par exemple, ou d'un formulaire) peuvent avoir ou ne pas avoir de valeur. Spécifier un champs nullable permet de savoir immédiatement si l'absence de valeurs est autorisé pour cette variable.

Pour déclarer une variable `a` de type `T` comme `nullable`, on peut soit écrire `T? a;` ou `Nullable<T> a;`. Cette variable peut être initialisée à `null` ou à toute valeur que prend le type `T`, par exemple `bool? b=null;` ou `int?  i=42;`.

Pour savoir si une valeur nullable a une valeur (non `null`) on utilise la méthode `HasValue()`. Pour récupérer la valeur contenue dans une telle variable nullable, soit on utilise le champ public `Value` (par exemple `int j = i.Value;`), soit on convertit cette variable nullable en variable non nullable (en utilisant l'opérateur de coalescence nulle `int j = i ??  0;` qui, ici, affectera la valeur 0 à j si i est null). Ci-dessous, un exemple dans lequel on simule un formulaire, la variable nullable ageInput n'ayant aucune valeur tant qu'elle n'est pas saisie :

```
public class AgeInput {

    private int? ageForm = null;

    public void set() {
        ageForm = Convert.ToInt32(Console.ReadLine());
    }
```

```
8
9       public int? get() {
10          if (!ageForm.HasValue)
11              Console.WriteLine("Aucune valeur d'age");
12          return ageForm;
13      }
14  }
```

... et son usage dans la fonction Main :

```
1       AgeInput ageInput = new AgeInput();
2       Console.WriteLine("Age = "+ageInput.get());
3       ageInput.set();
4       Console.WriteLine("Age = "+ageInput.get());
5       int age = ageInput.get().Value;
6       Console.WriteLine("Age = "+age);
7       age = ageInput.get() ?? 0;
8       Console.WriteLine("Age = "+age);
```

4.c Namespaces

Le système de namespaces de C# est exactement le même que celui de C++.

Les classes définies dans les API et celles que vous définissez appartiennent à un namespace (un espace de nom) qui lui même peut appartenir à des namespaces plus vastes et ainsi de suite. Dans l'exemple ci-dessous décrivant divers objets et créatures d'un jeu vidéo, ont voit que le découpage en namespaces permet de clarifier le code.

```
1   namespace jeuNS {
2
3       public abstract class ObjetJeu { /*...*/ }
4
5       namespace creaturesNS {
6           public abstract class Animal : ObjetJeu{ /*...*/}
7           public abstract class AnimalTerrestre : Animal { /*...*/}
8           public abstract class AnimalVolant : Animal { /*...*/}
9           public class Vache : AnimalTerrestre { /*...*/}
10          public class ChauveSouris : AnimalVolant { /*...*/}
11      }
12
13      namespace objetsNS {
14          public abstract class ObjetInerte : ObjetJeu { /*...*/ }
15          public abstract class Machine : ObjetJeu { /*...*/ }
16          public class Mixeur : Machine { /*...*/ }
17          public class Vase : ObjetInerte { /*...*/ }
18      }
19
20  }
```

4.d Classes, structs, records, ...

Types références ou valeurs

Le langage C# possède une grande richesse de types pouvant être définis. Mais ce qu'il faut retenir d'abord, c'est que .Net contient 2 grandes catégories de types : les `reference types` et les `value types`. Les classes (`class`) sont des `reference types` tandis que les `struct` sont des `value types`. Qu'est ce que cela veut dire ? En fait, les `reference types` vivent dans le tas (la partie dynamique de la mémoire ou de gros objets peuvent être instanciés) tandis que les `value types` vivent dans la partie statique de la mémoire, la pile, où les accès sont plus rapides mais où la taille instanciable est limitée. De plus, les `value types` contiennent forcément au moins une valeur, tandis que les `reference types` n'en contiennent pas forcément car ils peuvent ne contenir eux-même que des références nulles. Du point de vue de l'instanciation, les `reference types` sont implémentés comme des pointeurs vers les objets instanciés dans le tas.

Autrement dit, C# vous permet d'optimiser un peu la mémoire selon vos besoins.

Classes

Décrivons maintenant un peu ces types :

On retrouve les `class` sur un modèle équivalent à Java. Je ne reviens pas sur la possibilité de créer des `interface` et `abstract class`. Il est possible de préciser la non-héritabilité d'une classe avec le mot clef `sealed` (voir l'exemple des distances), équivalent du mot clef *final* de Java. Vous retrouverez également les mots clefs spécifiques au polymorphisme comme `override`. Bien entendu, vour retrouverez aussi le mot clef `static` qui permet de créer des classes non instanciables, ou de spécifier des variables dites "de classe". Tout cela fonctionne à peu près comme en Java.

Concernant la construction, il y a des subtilités que ne permet pas Java mais qui dépassent ce cours. La chose à savoir cependant, c'est que pareillement à Java, toutes les classes sont dérivées d'un type de base de C#, `System.Object`.

En plus de ce que permet Java, vous trouverez l'équivalent des destructeurs de C++, les `finalizers`. Il s'agit de méthodes particulières qui sont l'exact inverse de ce que font les constructeurs : ils indiquent comment détruire l'objet que vous avez instancié (`this`) et que faire à sa destruction. Leurs particularités sont qu'il n' peut y avoir qu'un seul finaliseur par classe, qu'ils ne peuvent être hérités, que leur appel n'est pas explicite (ils sont appelés à la destruction de l'objet), et qu'il n'ont ni modificateurs de portée ou autres, ni paramètres. Par exemple :

```
namespace DataTreatment {

    public class Data {
            private double[,] _Data { get; set; }
            public double[,] get() {return _Data;}
            public Data(int row, int col) {_Data = new double[row, col];}
            ~Data(){ Console.WriteLine("Destruction des données"); }
    }

}
```

Struct

En plus de cela, C# permet de définir un type d'objet supplémentaire et digne d'intérêt votre niveau, les `struct` (qui existent en C++).

Les `struct` peuvent être vues comme de petites classes contenant peu de valeurs et peu de méthodes. On définit une struct lorsqu'on met l'accent sur les données et pas le comportement d'un objet. Ce sont des `value types`. A la différences des `class` qui sont instanciées dans le `tas` et accessibles par référence, les `struct` sont instanciées dans la pile et accessibles par valeur. Les structs ne peuvent pas être déclarées `static`. Les structs ne peuvent pas hériter ; elles peuvent par contre implémenter des interfaces. Voici un exemple :

```
/* une classe de coordonnées immuables */
public readonly struct Coord(double x, double y) {
    public double X { init; get; } = x;
    public double Y { init; get; } = y;

    public Coord Translate(Coord c) {
        return new Coord(X + c.X, Y + c.Y);
    }

    public override String ToString() {
        return "(" + X + "," + Y + ")";
    }
}


internal class Program {
```

```
17
18      public static void Main(string[] args) {
19          Coord c1 = new Coord(3, 4);
20          Coord c2 = default;
21          c2 = c1.Translate(new Coord(1, -1));
22          Console.WriteLine("\n\nc1 translated : "+c1.Translate(new Coord(1,-1)))
                ;
23          Console.WriteLine("c2 : "+c2);
24      }
```

Notez que l'usage de `new` ne signifie en rien que l'allocation se fait dans le tas. L'opérateur `new` de C# est décliné sous différentes formes et peut allouer dans diverses parties de la mémoire et de différentes manières, selon l'objet à allouer.

Quel est l'intérêt de faire des structs ? Il y a au moins 4 points qui présentent un intérêt dans l'usage des structs plutôt que de faire des objets complexes:

- une struct est dédiée à des objets peu complexes avec peu de valeurs. Créer de tels objets augmente la lisibilité du code.
- les structs contenant peu de valeurs (petite taille en mémoire) et étant chargée dans la pile, il est intelligent de faire des `structs immuables`, c'est à dire dans lesquelles les valeurs ne peuvent être modifiées. Pour changer les valeurs, il faut allouer de nouveaux objets utilisant les valeurs des anciens. Nous verrons l'immuabilité dans la partie *programmation fonctionnelle* du cours.
- une struct étant un petit objet contenant peu de valeurs, l'intégralité de l'objet peut se trouver être chargé en mémoire cache, ce qui constitue une optimisation.
- ce sont aussi les vitesses d'accès qui changent. La pile est une partie de la mémoire très rapide d'accès de part sa structure. Choisir de faire des structs (des `value types`) permet d'optimiser les vitesses de traitement d'un programme.

class records et struct records

Tout d'abord, je vous enjoins à voir la documentation officielle qui dispose d'un très bon exemple sur les températures.

Parlons maintenant des enregistrements (`records`). Les enregistrements (`record`) sont un type qu'on peut créer à la place d'une classe ou d'une struct. Il existe ainsi des `record` (des types références), des `record struct` (qui sont des types valeur) et des `record class` (qui sont des types références).

Les enregistrements sont encore plus orientés données que ne le sont les structs. Ils permettent une écriture très compacte des données. Leur usage est fondé sur la donnée et non sur la référence. Cela signifie qu'on peut directement tester l'égalité des objets d'enregistrement entre eux ; dans ce cas, ce qui est testé n'est pas l'égalité des références mais l'égalité des données contenues : deux instances de record sont égales si les valeurs de toutes leurs propriétés et champs sont égales. Ils sont donc a considérer particulièrement dans des usages fortements orientés données ou statistiques.

Les enregistrements disposent ainsi de méthodes permettant de tester leur égalité ou leur différence (les opérateurs `==` et `!=`, mais aussi des méthodes `Object.Equals(Object)` et `Object.GetHashCode()` modifiées pour vérifier l'égalité des attributs (valeurs).

Les enregistrements possèdent une méthode `ToString()` affichant d'emblée les valeurs et les noms des membres.

Il est possible de définir des propriétés (que nous verrons dans la section suivante) dans les enregistrements.

Comme les classes, les records sont héritables et peuvent faire l'objet d'abstractions.

Voici un exemple d'usage complet utilisant des types `record struct`. Pour commencer voici ci-dessous le record `Biometrie` qui contient des données de poids, de taille et de sexe, mais aussi l'indice de masse corporelle ($IMC = poids/taille^2$) :

```
1 public enum Sexe { F, H }
2 public enum IMCCateg {Denutrition,Maigreur,Normal,Surpoids,ObesiteModeree,
      ObesiteSevere,ObesiteMorbide }
```

```
3  public enum Comparaison { Inferieur, Egal, Superieur }
4
5  public readonly record struct Biometrie(Sexe sexe, double taille, double poids)
       {
6      private static double[] IMCBreaks = { 16.5, 18.5, 25, 30, 35, 40 };
7
8      // calcule l'IMC comme une propriété
9      public double IMC => poids/Math.Pow(taille/100, 2);
10
11     // retourne la catégorie d'IMC
12     public IMCCateg getIMCCategorie() {
13         int index = 0;
14         for (int i = 0; i < IMCBreaks.Length; i++) {
15             if (IMC > IMCBreaks[i])
16                 index++;
17         }
18         IMCCateg t = (IMCCateg)(Enum.GetValues<IMCCateg>()).GetValue(index);
19         return t;
20     }
21
22     private int getIndex(IMCCateg c) {
23         // récupere l'index de la catégorie en paramètre dans l'enum
24         int index = Array.IndexOf(Enum.GetValues(c.GetType()), c);
25         return index;
26     }
27
28     // retourne la catégorie d'IMC
29     public Comparaison CompareTo(IMCCateg refIMC) {
30         // récupere l'index de la catégorie en paramètre dans l'enum
31         // et compare l'IMC avec le seuil
32         int index = getIndex(refIMC);
33         if (index>IMCBreaks.Length-1) return Comparaison.Inferieur;
34         if (IMC > IMCBreaks[index]) return Comparaison.Superieur;
35         if (refIMC == getIMCCategorie()) return Comparaison.Egal;
36         return Comparaison.Inferieur;
37     }
38
39 }
```

Ce record est une struct et permet ainsi d'être `readonly`, donc immuable. On définit 3 énumérations, une pour définir le sexe, une autre pour définir si une comparaison est supérieure, inférieure ou égale, une dernière dans laquelle sont listées toutes les catégories d'indices de masse corporelle. NB: les enumérations ont été définies à l'extérieur du record pour simplifier leur accès de l'extérieur et éviter de devoir écrire `Biometrie.Sexe.F` au lieu de `Sese.F`.

Le record contient un tableau de valeurs fixes qui sont les seuils permettant de passer d'une catégorie d'IMC à une autre. Vous constaterez que les variables (immuables) contenant les valeurs de poids, de taille et de sexe, sont définies uniquement au tout début de la déclaration du record. C'est une forme d'écriture nommée constructeur primaire.

En plus de cela, le record contient une propriété nommée IMC dans laquelle on trouve, sous la forme d'une écriture fonctionnelle, son calcul à partir du poids et de la taille.

On voit aussi que le record contient 3 fonctions. Une fonction `IMCCateg getIMCCategorie()` qui permet de savoir à quelle catégorie d'IMC appartient la personne. Une fonction privée `int getIndex(IMCCateg c)` qui, étant donné une catégorie récupère son index dans l'énumération. Une fonction `Comparaison CompareTo(IMCCateg refIMC)` qui renvoie une comparaison permettant de savoir si la personne concernée est dans une catégorie égale, inférieure ou supérieure à celle passée en paramètre.

Ajoutons d'autres records. Dans l'exemple qui suit, vous voyez qu'un enregistrement abstrait est définit. Il contient une liste (`IEnumerable`) d'enregistrements Biometrie. Les deux autres records héritent de ce premier enregistrement. Ces deux autres records sont spécialisés : ils réalisent de manière fonctionnelle des comptages de biométries répondant à certaines conditions. Ne vous attardez pas ici sur le code ; nous verrons cela plus tard dans la partie programmation fonctionnelle ; vous constaterez seulement que ce code fait appel aux fonctions définies dans le record Biometrie.

Le record `SumIMCOfCategC(IMCCateg Categ, IEnumerable<Biometrie> BmRecords)` compte combien de biométries correspondent à la catégorie passée en paramètre. Le record `SumIMCComparedToCategC(IMCCateg Categ, Comparaison comp, IEnumerable<Biometrie> BmRecords)` compte combien de biométries satisfont la comparaison avec la catégorie passées en paramètre. Vous constaterez que ces 2 records sont `sealed` ; ils ne peuvent plus être hérités. Vous constaterez aussi l'écriture très compacte de ces traitements.

```
public abstract record ListOfBiometries(IEnumerable<Biometrie> BMRecords);

public sealed record SumIMCOfCategC(IMCCateg Categ, IEnumerable<Biometrie>
    BmRecords)
    : ListOfBiometries(BmRecords) {
    public double IMCs => BMRecords.Where(s => s.getIMCCategorie()==Categ).Sum(
        s => Convert.ToInt32(s.getIMCCategorie()==Categ));
}

public sealed record SumIMCComparedToCategC(IMCCateg Categ, Comparaison comp,
    IEnumerable<Biometrie> BmRecords)
    : ListOfBiometries(BmRecords) {
    public double IMCs => BMRecords.Where(s => s.CompareTo(Categ)==comp).Sum(s
        => Convert.ToInt32(s.CompareTo(Categ)==comp));
}
```

Vous trouverez ci-dessous la classe `Program` qui contient la fonction `Main`, et en dessous encore la sortie console. Dans cette classe on déclare et initialise un tableau statique de valeurs biométriques.

La fonction main contient d'une part l'affichage des biométries, d'autre part l'usage de traitements utilisant des records spécialisés hérités du record `ListOfBiometries`. Vous constaterez l'écriture très compacte mais très claire, le fait que la fonction `ToString` par défaut des records fait le travail, et également le fait de définir un enregistrement abstrait permet de stocker ces enregistrements dans une liste d'instances de records (`List<ListOfBiometries> treatments`) qui peuvent tous être appelés dans une boucle.

```
internal class Program {

    private static Biometrie[] _biodata =
    [ new Biometrie(Sexe.H, 180,73),
      new Biometrie(Sexe.F, 168,58),
      new Biometrie(Sexe.H, 186,69),
      new Biometrie(Sexe.H, 178,73),
      new Biometrie(Sexe.F, 155,72),
      new Biometrie(Sexe.F, 175,65),
      new Biometrie(Sexe.H, 192,84)
    ];

    public static void Main(string[] args) {
        Console.WriteLine("--- Affichage de l'ensembles des biométries ---");
        foreach (var item in _biodata)
            Console.WriteLine(item+" -> "+item.getIMCCategorie());

        Console.WriteLine("\n--- succession de traitements utilisant des
            records ---");
        List<ListOfBiometries> treatments = new List<ListOfBiometries>();
        treatments.Add(new SumIMCOfCategC(IMCCateg.Normal, _biodata));
        treatments.Add(new SumIMCOfCategC(IMCCateg.Surpoids, _biodata));
        treatments.Add(new SumIMCComparedToCategC(IMCCateg.Normal, Comparaison.
            Inferieur, _biodata));
        treatments.Add(new SumIMCComparedToCategC(IMCCateg.Surpoids,
            Comparaison.Inferieur, _biodata));
        treatments.Add(new SumIMCComparedToCategC(IMCCateg.Surpoids,
            Comparaison.Superieur, _biodata));
        foreach (var item in treatments)
            Console.WriteLine(item);
```

```
    }
}
```

```
--- Affichage de l'ensembles des biométries ---
Biometrie { sexe = H, taille = 180, poids = 73, IMC = 22,530864197530864 } ->
    Normal
Biometrie { sexe = F, taille = 168, poids = 58, IMC = 20,549886621315196 } ->
    Normal
Biometrie { sexe = H, taille = 186, poids = 69, IMC = 19,944502254595903 } ->
    Normal
Biometrie { sexe = H, taille = 181, poids = 52, IMC = 15,872531363511492 } ->
    Denutrition
Biometrie { sexe = H, taille = 178, poids = 73, IMC = 23,04002019946976 } ->
    Normal
Biometrie { sexe = F, taille = 155, poids = 72, IMC = 29,968782518210194 } ->
    Surpoids
Biometrie { sexe = F, taille = 175, poids = 65, IMC = 21,224489795918366 } ->
    Normal
Biometrie { sexe = H, taille = 192, poids = 84, IMC = 22,786458333333336 } ->
    Normal
Biometrie { sexe = H, taille = 172, poids = 105, IMC = 35,49215792320173 } ->
    ObesiteSevere

--- succession de traitements utilisant des records ---
SumIMCOfCategC { BMRecords = DataTreatment.Biometrie[], Categ = Normal,
    BmRecords = DataTreatment.Biometrie[], IMCs = 6 }
SumIMCOfCategC { BMRecords = DataTreatment.Biometrie[], Categ = Surpoids,
    BmRecords = DataTreatment.Biometrie[], IMCs = 1 }
SumIMCComparedToCategC { BMRecords = DataTreatment.Biometrie[], Categ = Normal,
     comp = Inferieur, BmRecords = DataTreatment.Biometrie[], IMCs = 1 }
SumIMCComparedToCategC { BMRecords = DataTreatment.Biometrie[], Categ =
    Surpoids, comp = Inferieur, BmRecords = DataTreatment.Biometrie[], IMCs = 7
    }
SumIMCComparedToCategC { BMRecords = DataTreatment.Biometrie[], Categ =
    Surpoids, comp = Superieur, BmRecords = DataTreatment.Biometrie[], IMCs = 1
    }
```

4.e Variables, Constantes, Propriétés, Indexeurs

Dans cette section, par "variable", je désigne aussi bien les variables déclarées dans une portée (entre accolades) que les variables de classe (ou struct, records, ..., autrement dit les attributs).

Il y a peu à dire qui diffère de Java ou C++ sur les variables. Ce qui est important de retenir à leur propos en C# est qu'elles peuvent être déclarées immuables ! Nous reviendrons sur ce point mais je vais ici tout de suite introduire la différence entre constantes (`const`) et variables immuables (`readonly`). Je vais aussi parler des combinaisons possibles avec le modificateur `static`.

Variables et constantes - immuabilité / statique

Une constante est une zone de mémoire accessible par une variable nommée initialisée à la compilation ; sa valeur est inscrite en dur dans le code machine à chaque endroit où elle est utilisée ; elle ne peut être modifiée. Une variable immuable est une variable qui ne peut pas être modifiée non plus, mais à la différence d'une constante, elle n'est pas créée (réservée en mémoire) à la compilation, mais à l'execution, par contre, une fois instanciée et initialisée, elle ne peut plus être modifiée. Nous verrons leur utilité plus tard. Pour rappel, une variable déclarée `static` est une variable dite "de classe", c'est-à-dire que sa valeur n'est pas spécifique d'une instance particulière ; si une instance d'une classe contenant un champ statique modifie ce dernier, alors cette modification sera valble pour toutes les instances de cette classe.

Voyons un exemple des possibilités que l'on a en termes de variables et constantes :

```
public record Point {
    private int _x;
```

```
3       private int _y;
4
5       public Point(int x, int y) {
6       set(x,y);
7       }
8
9       public void set(int x, int y) {
10          _x = x;
11          _y = y;
12      }
13  }
14
15  public class Variables_Constants {
16
17      // Constantes
18      private const int _A_CONSTANT = 42;
19
20      // Variables statiques ("de classe")
21      private static int _initializedStaticVariable = 24;
22      private static int _staticVariable;
23
24      // Variable mutable
25      private int _mutableVariable;
26
27      // Variables immuables
28      private readonly int _initializedImutableVariable = 36;
29      private readonly int _immutableVariable;
30
31      // Variables immuables statiques ("de classe")
32      private static readonly int _initializedStaticImmutableVariable = 42;
33      private static readonly int _staticImmutableVariable;
34
35      // type reference static readonly
36      public static readonly Point point = new Point( 15, 5);
37
38
39      // Constructeur
40      public Variables_Constants(int p) {
41          // _A_CONSTANT = p; // --> Erreur, non autorisé
42          _initializedStaticVariable = p; // ok, on peut réinitialiser la valeur
                à la construction
43          _staticVariable = p; // ok, on peut initialiser la valeur à la
                construction
44          _mutableVariable = p; // ok, on peut initialiser la valeur à la
                construction
45          _initializedImutableVariable = p; // ok, on peut réinitialiser la
                valeur à la construction
46          _immutableVariable = p; // ok, on peut réinitialiser la valeur à la
                construction
47          // _initializedStaticImmutableVariable = p;  // --> Erreur, non autoris
                é
48          // _staticImmutableVariable = p;  // --> Erreur, non autorisé
49          point.set(p,p); // ok, on peut réinitialiser les champs internes la
                construction
50      }
51
52      // Mutateur
53      public void Set(int p) {
54          // _A_CONSTANT = p; // --> Erreur, non autorisé
55          _initializedStaticVariable = p; // ok, on peut changer la valeur
56          _staticVariable = p; // ok, on peut changer la valeur
57          _mutableVariable = p; // ok, on peut changer la valeur
58          // _initializedImutableVariable = p; // --> Erreur, non autorisé, la
```

```
            variable est immuable
59          // _immutableVariable = p; // --> Erreur, non autorisé, la variable est
                immuable
60          // _initializedStaticImmutableVariable = p;  // --> Erreur, non autoris
                é
61          // _staticImmutableVariable = p;  // --> Erreur, non autorisé
62          point.set(p,p); // ok, on peut modifier les champs internes
63      }
64
65      public override String ToString() {
66          return "\n_A_CONSTANT = "+_A_CONSTANT+"\n"+
67                 "_initializedStaticVariable = "+ _initializedStaticVariable+"\n"
                    +
68                 "_staticVariable = "+ _staticVariable + "\n" +
69                 "_mutableVariable = "+ _mutableVariable + "\n" +
70                 "_initializedImutableVariable = "+ _initializedImutableVariable
                    + "\n" +
71                 "_immutableVariable = "+ _immutableVariable + "\n" +
72                 "_initializedStaticImmutableVariable = "+
                        _initializedStaticImmutableVariable + "\n" +
73                 "_staticImmutableVariable = "+ _staticImmutableVariable+ "\n" +
74                 point;
75      }
76  }
```

Notez que par convention les constantes s'écrivent habituellement en MAJUSCULES.

le code appelant (dans la fonction Main) :

```
1       // les champs initialisables à la construction doivent prendre la valeur 1
2       Variables_Constants vc = new Variables_Constants(1);
3       Console.WriteLine(vc);
4       // les champs modifiables (mutables) doivent prendre la valeur 2
5       vc.Set(2);
6       Console.WriteLine(vc);
```

et la sortie console :

```
1   _A_CONSTANT = 42
2   _initializedStaticVariable = 1
3   _staticVariable = 1
4   _mutableVariable = 1
5   _initializedImutableVariable = 1
6   _immutableVariable = 1
7   _initializedStaticImmutableVariable = 42
8   _staticImmutableVariable = 0
9   Point { _x = 1, _y = 1 }
10
11  _A_CONSTANT = 42
12  _initializedStaticVariable = 2
13  _staticVariable = 2
14  _mutableVariable = 2
15  _initializedImutableVariable = 1
16  _immutableVariable = 1
17  _initializedStaticImmutableVariable = 42
18  _staticImmutableVariable = 0
19  Point { _x = 2, _y = 2 }
```

Dans le code, on voit que :

- les constantes (`const`) doivent être initialisées directement à leur déclaration.
- les variables, qu'elles soient `statiques mutables` ou `mutables` ou `immuables` (le 'ou' étant exclusif dans cette phrase), peuvent être initialisées à la construction.

- les variables immuables ne peuvent plus être modifiées après leur initialisation à la construction.
- les variables `statiques immuables` (`static readonly`) ne peuvent ni être initialisées à la construction, ni modifiées par la suite.
- la variable de type référence (un `record) point` est statique est immuable (donc on ne peut pas changer la référence) mais ses champs internes sont mutables. Pour qu'ils ne le soient pas, il faudrait qu'ils soient eux-mêmes immuables (ce qui est possible en définissant le record comme `readonly struct record`.

On peut faire un parallèle entre les `const` ou `static readonly` : dans les deux cas, elles ne peuvent être ni initialisées à la construction, ni modifiées. Alors, quelle différence ? Si les valeurs des constantes sont inscrites en dur dans le code machine à la compilation à chaque endroit où elles sont utilisées, ce n'est pas le cas des variables déclarées `static readonly`. Les valeurs de ces dernières sont inscrites à un seul endroit dans le code qui s'y réfère. Dès lors, que choisir entre les deux ? Hormis un faible gain de vitesse à l'exécution avec l'usage de constantes, l'avantage revient aux variables statiques immuables pour plusieurs raisons, l'usage des constantes devant être limité aux valeurs fréquemment utilisées comme π :

- si, dans un programme, on décide de changer la valeur d'une constante, on est obligé de recompiler tout le projet (qui peut être gros).
- si la variable est de type référence, il est tout à fait possible de l'allouer dynamiquement dans le tas avec l'opérateur `new`, ce qui n'est pas possible avec une constante : `public static readonly Point p = new Point  X = 15, Y = 5 ;`. Mais c'est dans ce cas la référence à l'objet qui est constante (voir l'exemple plus haut). Ses champs sont modifiables. Cela permet donc de créer des objets instanciés de façon immuable et statique (de classe).
- il est possible d'instancier des objets d'une classe donnée dans cette même classe et de s'en servir comme valeurs de référence en les rendant valables pout l'ensemble des instances (`static`) et en empêchant les utilisateurs de modifier ces valeurs (`readonly`). Cet exemple (ci-dessous) tiré de stackoverflow montre un usage intéressant de ce dernier point.

```
1  public class Color {
2      public static readonly Color Black = new Color(0, 0, 0);
3      public static readonly Color White = new Color(255, 255, 255);
4      public static readonly Color Red   = new Color(255, 0, 0);
5      public static readonly Color Green = new Color(0, 255, 0);
6      public static readonly Color Blue  = new Color(0, 0, 255);
7      private byte red, green, blue;
8
9      public Color(byte r, byte g, byte b) => (red, green, blue) = (r, g, b);
10 }
```

Propriétés

Une propriété est un champ (un attribut) au même titre qu'une variable ou qu'une constante, mais ces champs comportent en plus de cela des accesseurs (getters/setters) définissant comment ces valeurs sont créées ou modifiées, et comment on y accède.

Une propriété possède soit un initialisateur `init` et un accesseur `get`, soit un mutateur `set` et un accesseur `get`. Dans le premier cas (couple `init` et `get`), la propriété ne peut qu'être initialisée soit à la construction, soit plus tard, mais pas modifiée. Dans le second cas, la propriété est mutable (modificable). Ces accesseurs peuvent être publics ou privés.

Voici un exemple de propriété écrite de façon très compacte ; elle n'explicite même pas les accesseurs qui sont définis par défaut :

```
1  public readonly record struct Vector(double x, double y) {
2      public double Norm => Math.Sqrt(x * x + y * y);
3      public bool IsNormalized { get; init; } = Math.Abs(Math.Sqrt(x * x + y * y)
           - 1) < 1e-10;
4  }
```

Dans cet exemple, on définit 2 champs, x et y (de type valeurs) ainsi que 2 propriétés. On voit que la propriété `IsNormalized` est uniquement initialisable. Le record possède donc ici 4 champs. Il n'y a pas de constructeurs dans le corps de l'enregistrement ; l'enregistrement est ici écrit dans la forme `constructeur primaire`.

Voici un autre exemple plus complexe :

```
public readonly record struct Vector {
    public double X { get; }
    public double Y { get; }
    public double Norm { get; }
    public bool IsNormalized { get; }
    public double Angle { get; } // Angle in radians

    public Vector(double x, double y) : this(x, y, Math.Sqrt(x * x + y * y)) {}

    private Vector(double x, double y, double norm) {
        X = x;
        Y = y;
        Norm = norm;
        IsNormalized = Math.Abs(norm - 1) < 1e-10;
        Angle = Math.Atan2(y, x)*180/Math.PI; // Calculates the angle in
            radians
    }
    public override string ToString() =>
        $"Vector(x: {X}, y: {Y}, Norm: {Norm}, IsNormalized: {IsNormalized},
            Angle: {Angle})";

    public Vector Translate(Vector v) {
        return new Vector(X + v.X, Y + v.Y);
    }

    public Vector Rotate(double angle) {
        double angle_rad = angle * Math.PI / 180;
        double x = Math.Round(X * Math.Cos(angle_rad)+Y * Math.Sin(angle_rad));
        double y = Math.Round(-X * Math.Sin(angle_rad) + Y * Math.Cos(angle_rad
            ));
        return new Vector(x, y);
    }
}
```

Dans cet exemple, ont voit que tous les champs sont des propriétés (`X`, `Y`, `Norm`, `IsNormalized` et `Angle`). Elles sont toutes définies seulement avec l'accesseur `get`, l'initialisateur étant implicite ici. On voit aussi que des méthodes (`Translate` et `Rotate`) ont été ajoutées. On remarque surtout qu'il y a 2 constructeurs, un public qui permet de saisir les valeurs des composantes `X` et `Y`, et l'autre dans lequel les calculs des propriétés sont effectués. La raison est que sinon les propriétés, dans le corps de la fonction, peuvent être calculées à partir des valeurs fournies à la construction mais pas à partir de valeurs d'autres propriétés.

Indexeurs

Les indexeurs sont très semblables aux propriétés à ceci près qu'ils peuvent prendre un paramètre. Ils servent à accéder de manière indexée aux champs membres d'une classe. Ils sont utiles dans le traitement automatique des données, quand on n'a pas particulièrement besoin de s'intéresser aux champs membres en soit. Un exemple vaut mieux qu'un long discours. Supposons une fiche d'identité d'une personne :

```
namespace DataTreatment;

public record Individu(String nom, String prenom, Sexe sexe, int age, int
    taille, double poids) {
    public String Nom { get; init; } = nom;
    public String Prenom { get; init; } = prenom;
    public Sexe Sexe { get; init; } = sexe;
```

```
    public int Age { get; set; } = age;
    public int Taille { get; set; } = taille;
    public double Poids { get; set; } = poids;

    // indexeur utilisant des indices entiers
    public object this[int index] {
        get {
            if (index == 0)
                return Nom;
            else if (index == 1)
                return Prenom;
            else if (index == 2)
                return Sexe;
            else if (index == 3)
                return Age;
            else if (index == 4)
                return Taille;
            else if (index == 5)
                return Poids;
            else
                return null;
        }
        set {
            if (index == 3)
                Age = Convert.ToInt32(value);
            else if (index == 4)
                Taille = Convert.ToInt32(value);
            else if (index == 5)
                Poids = Convert.ToDouble(value);
        }
    }
}
```

L'indexeur renvoie ici un type générique (`object`) car les différents champs (sexe, taille, nom...) sont de types différents. On voit qu'il est construit autour de `this` et d'un paramètre (ici un index de type entier) passé entre crochets.

Cela permet d'écrire ceci :

```
Individu indiv = new Individu("Glade", "Nicolas", Sexe.H, 48, 180, 73);
for(int i=0; i<6; i++)
    Console.WriteLine(indiv[i]);
```

On accède ici aux champs successifs via un index.

Cela peut s'avérer très pratique dans des traitements automatisés de données. Cet exemple souffre cependant d'un défaut de conception. On voit que les propriétés `Nom`, `Prenom` et `Sexe` sont immuables (seulement initialisables), donc le setter (`set`) de l'indexeur ne commence qu'à 3 (pour l'age). Ce n'est pas très élégant. On pourrait remplacer (ou rajouter) l'indexeur par un indexeur fondé sur une chaine de caractère comme suit :

```
    public object this[String index] {
        get {
            if (index == "nom")
                return Nom;
            else if (index == "prenom")
                return Prenom;
            else if (index == "sexe")
                return Sexe;
            else if (index == "age")
                return Age;
            else if (index == "taille")
                return Taille;
            else if (index == "poids")
                return Poids;
            else
```

```
                return null;
            }
            set {
                if (index == "age")
                    Age = Convert.ToInt32(value);
                else if (index == "taille")
                    Taille = Convert.ToInt32(value);
                else if (index == "poids")
                    Poids = Convert.ToDouble(value);
            }
        }
```

Ce qui permet de faire ceci :

```
String[] indiv_fields = { "nom", "prenom", "sexe", "age", "taille", "poids" };
for (int i = 0; i < 6; i++)
    Console.WriteLine(indiv[indiv_fields[i]]);
// ou ...
Console.WriteLine("IMC = "+((double)indiv["poids"])*10000/Math.Pow((int)indiv["
    taille"],2));
```

4.f Iterateurs

Le framework .Net comporte un ensemble de design patterns absolument indispendables : ils permettent de rendre des objets itérables, autrement dit il permettent l'usage de `foreach` pour itérer dans des collections. Il s'agit de `IEnumerable<T>` et `IEnumerator<T>`. Il s'agit de 2 interfaces que toutes les structures de données (tableaux compris) de C# implémentent. Un objet `IEnumerable` se comporte comme une collection.

Definition. Un itérateur est une methode, un opérateur ou un getter qui comporte les caractéristiques suivantes : (1) le type de retour est `IEnumerable`, `IEnumerator`, `IEnumerable<T>` ou `Ienumerator<T>`, (2) le corps de la méthode / opérateur / getter contient le mot clef `yield` ("récolte").

Un exemple ? L'exemple qui suit crée un `IEnumerable<int>` contenant tous les nombres premiers entre 1 et n, calculés par l'algorithme du crible d'Eratosthène :

```
public class Primes {

    /* Implémente le crible d'Eratosthène pour calculer les nombres premiers
    * inférieurs ou égaux à n
    */
    public static IEnumerable<int> GetPrimesCE(int n) {
        if (n < 2) yield break; // Il n'y a pas de nombres premiers en dessous
            de 2
        // Créer et initialiser un tablea de bool, tous les nombres sont
            potentiellement premiers
        bool[] isPrime = new bool[n + 1];
        for (int i = 2; i <= n; i++) {
            isPrime[i] = true;
        }
        // Eliminer les multiples de i
        for (int i = 2; i * i <= n; i++) {
            if (isPrime[i]) {
                for (int j = i * i; j <= n; j += i) {
                    isPrime[j] = false;
                }
            }
        }
        // Renvoyer les nombres premiers
        for (int i = 2; i <= n; i++) {
            if (isPrime[i]) {
                yield return i;
            }
```

```
        }
    }
}
```

et usage :

```
foreach (var prime in Primes.GetPrimesCE(30)) {
    Console.WriteLine(prime);
}
```

Déjà, on voit à l'usage que l'on peut itérer avec `foreach` dans ce ce que renvoie la méthode `GetPrimesCE`. Nous verrons plus tard, dans le cours de programmation fonctionnelle, lorsque nous utiliserons LINQ, que ceci est indispensable.

Que fait `yield` ? Le mot clef `yield` se place toujours avant `break` ou `return` ? Yield recolte des valeurs lorsqu'il est placé avant return, et ces valeurs font partie du retour sous forme d'enumerable ou d'énumérateur. Mais attention, contrairement à une fonction normale qui, dès qu'elle rencontre `return` stoppe son execution et renvoie la valeur, cette fois la méthode n'est pas arrêtée tant que l'algorithme n'est pas terminé et la collection constituée. En réalité, un bloc itérateur (une méthode, un opérateur qui renvoie un enumérable ou un énumérateur) n'est pas executé. Il constitue un bloc de code qui est interprété par le compilateur de manière à générer un code qui récolte la collection.

Le type de retour (le `yield type`) est un `object` lorsque le bloc itérateur renvoie un `IEnumerable` ou un `IEnumerator` ; c'est un type `T` lorsque le bloc itérateur retourne un `IEnumerable<T>` ou un`Ienumerator<T>`.

Si l'on revient à notre exemple, on voit que dès le départ, on teste n, car si n est plus petit que 2, il n'y a pas de nombres premiers ; dans ce cas, on sort de la fonction en executant `yield break`. Pour récolter les nombres premiers trouvés, on utilise `yield return i;`.

Notez qu'il peut y avoir plusieurs récoltes dans le même bloc itérateur, et que ce bloc itérateur peut sans problème s'appeller lui même (récursivité), donc utiliser la récolte connue. Cet autre exemple d'implémentation de recherche des nombres premiers illustre ces 2 points :

```
    // implémente l'algorithme de recherche de diviseurs
    // pour calculer les nombres premiers inférieurs ou égaux à n
    public static IEnumerable<int> GetPrimesRD(int n)  {
        if (n < 2) yield break; // Il n'y a pas de nombres premiers en dessous
            de 2
        // Le premier nombre premier est 2
        yield return 2;
        // Parcourir tous les nombres impairs de 3 à n
        for (int i = 3; i <= n; i += 2) {
            bool found = false;
            double sqrt_i = Math.Sqrt(i);
            // Chercher un diviseur parmi les nombres premiers déjà trouvés
            foreach (var prime in GetPrimesRD((int)sqrt_i)) {
                if (i % prime == 0) {
                    found = true; // i n'est pas premier
                    break;
                }
            }
            // Si aucun diviseur n'a été trouvé, i est premier
            if (!found) {
                yield return i;
            }
        }
    }
```

`IEnumerator` et `IEnumerable`. Un objet `IEnumerable` est un `IEnumerator` (héritage). Un `IEnumerator` dispose des méthodes pour itérer dans une collection (dans une seule direction). Il dispose de :

- `Current` : l'item courant de la collection.

- `MoveNext()` : avance au prochain item ; retourne faux si on a atteint la fin de la collection.
- `Reset()` : remet l'énumérateur à la position initiale (premier item de la collection)
- `Dispose()` : cette méthode est utilisée pour libérer les ressources non managées (comme les handles de fichiers, les connexions réseau, les objets COM, etc.) dans les classes implémentant l'interface `IDisposable`. Dans la plupart des scénarios courants en C#, les objets sont gérés par le ramasse-miettes (Garbage Collector), mais si un objet utilise des ressources non managées, il est nécessaire de les libérer explicitement. Dans ce cas, l'objet à libérer devra implémenter `IDisposable`.

Il est possible d'utiliser ces méthodes manuellement, mais ce n'est pas recommandé ; il vaut mieux utiliser `foreach` en implémentant `IEnumerable`.

Un `IEnumerable` permet de retrouver un `IEnumerator` dans une collection. Une classe implémentant un `IEnumerable` peut être parcourue avec `foreach`. Une classe implémentant `IEnumerable` doit contenir `IEnumerator<T> GetEnumerator()` et `IEnumerator GetEnumerator()` qui lui permet de récupérer une énumérateur.

Donc, pour résumer, le `IEnumerator` décrit comment itérer dans une collection et le `IEnumerable` permet d'itérer.

Voici un exemple complet qui montre comment :

- créer un `IEnumerator` personnalisé, ici un énumérateur qui lit uniquement les caractères diagonaux d'une matrice carrée (un tableau 2D).
- créer une collection personnalisée implémentant `IEnumerable`, qui permet d'extraire les coefficients diagonaux d'une matrice.

Tout d'abord l'énumérateur :

```
1   public class DiagonalEnumerator : IEnumerator<int> {
2       private int[][] coefs;
3       private int position = -1;
4       public DiagonalEnumerator(int[][] coefs) {
5           this.coefs = coefs;
6       }
7       public bool MoveNext() {
8           position++;
9           return position < coefs.Length;
10      }
11      public void Reset() {
12          position = -1;
13      }
14      object IEnumerator.Current => Current;
15      public int Current => coefs[position][position];
16      public void Dispose() { /* Rien à nettoyer dans ce cas */ }
17  }
```

On voit que l'énumérateur implémente `IEnumerator<int>`, donc contient une propriété `Current` de type `int`, le type de l'énumérateur, implémente les méthodes `MoveNext()`, `Reset()` et `Dispose()`. On voit que l'implémentation de `MoveNext()` fait en sorte de faire croitre un entier nommé `position` et que cela s'arrête lorsqu'on a atteint la taille maximale du tableau de coefficients (de la matrice). On voit aussi comment est définie la propriété `Current` ici : elle est telle qu'elle correspond uniquement aux coefficients diagonaux (`coefs[position][position]`).

Pour créer une collection personnalisée, ici une matrice dont on extrait les éléments diagonaux, on fait :

```
1   public class DiagonalCoefs : IEnumerable<int> {
2       private int[][] matrix;
3
4       // Constructeur qui accepte une matrice carrée
5       public DiagonalCoefs(int[][] matrix) {
6           // Vérification que la matrice est carrée
```

```
            if (matrix == null || matrix.Length == 0 || matrix.Length != matrix
                [0].Length) {
                throw new ArgumentException("La matrice doit être carrée.");
            }
            this.matrix = matrix;
        }

        // Implémentation de IEnumerable<int>
        public IEnumerator<int> GetEnumerator() {
            return new DiagonalEnumerator(matrix);
        }

        IEnumerator IEnumerable.GetEnumerator() {
            return GetEnumerator();
        }
    }
```

La classe `DiagonalCoefs`, une collection personnalisée, implémente `IEnumerable<int>` ce qui lui permet d'être parcourue par un `foreach`. On voit comment les methodes `GetEnumerator()` renvoient l'énumérateur personnalisé que nous avons défini `DiagonalEnumerator(matrix)`.

Et maintenant l'usage dans le Main :

```
    int[][] matrix = {
        new int[] { 1, 2, 3 },
        new int[] { 4, 5, 6 },
        new int[] { 7, 8, 9 }
        };

        DiagonalCoefs diagonalCoefs = new DiagonalCoefs(matrix);

        foreach (var value in diagonalCoefs) {
            Console.WriteLine(value);  // Affichera les valeurs de la diagonale
                 : 1, 5, 9
        }
```