Les Types valeurs et références. (Objets et structures, nullable)

Il existe deux genres de types en C# : les types référence et les types valeur. Les variables des types référence font référence à leurs données (objets), tandis que les variables des types valeur contiennent directement leurs données. Avec les types référence, deux variables peuvent faire référence au même objet ; par conséquent, les opérations sur une variable peuvent affecter le même objet référencé par l'autre variable. Avec les types valeur, chaque variable a sa propre copie des données et les opérations sur une variable ne peuvent pas affecter l’autre

Types valeur :

Il existe deux sortes de types valeur : Structures et énumérations.

Tous les types valeur sont implicitement dérivés de [System.ValueType](https://docs.microsoft.com/fr-fr/dotnet/api/system.valuetype).

Les types simples sont un ensemble de types struct prédéfinis fournis par C#, composé des types suivants :

Types intégraux : types numériques entiers et type char

Types virgule flottante

bool

Type référence :

Les mots clés suivants sont utilisés pour déclarer des types référence :

Classe / interface / delagate

C# fournit également les types référence intégrés suivants :

Dynamic / object / string

EXEMPLE DE CODE ENTRE UNE VALEUR TYPE VALEUR ET UNE TYPE REF

Les chaines de caractères et string builders

Une chaîne de caractère est un objet de type String dont la valeur est du texte. Il n’y a pas de caractère de fin NULL à la fin de la chaîne, elle peut donc en contenir.

Exemple pour initialiser un string :

string exemple = "chaine d’objet char";

String Builder :

Importation de l’espace de noms System.Text : using System.Text;

Class StringBuilder : on peut utiliser la classe la classe StringBuilder pour créer une chaine de caractères et indiquer sa capacité, si cette capacité vient à être dépassé, elle est automatiquement réallouée en la doublant.

StringBuilder myStringBuilder = new StringBuilder("Hello World!", 25);   
On peut indiquer une valeur maximale de capacité avec la propriété Capacity.  
On peut utiliser les méthodes append, insert, remove et replace.

On peut convertir un objet StringBuilder en String grâce à la méthode ToString

Boxing et unboxing

Le boxing est utilisé pour stocker des types valeur dans le tas rassemblé par garbage collection. Le boxing est une conversion implicite d’un type valeur en type object ou en un type interface implémenté par ce type valeur. Le boxing d'un type valeur alloue une instance d'objet sur le tas et copie la valeur dans le nouvel objet.

Dans l'exemple suivant, une variable de type valeur est déclarée :

int i = 123;

L'instruction ci-dessous réalise implicitement une opération de boxing sur la variable i :

// Boxing copies the value of i into object o.

object o = i;

int i = 123;

object o = (object)i; // explicit boxing

Le type valeur d'origine et que l'objet boxed utilisent des emplacements de mémoire distincts et peuvent, par conséquent, stocker des valeurs différentes.

L’unboxing est une conversion explicite du type object en un type valeur, ou d’un type interface en un type valeur qui implémente l’interface. Une opération d'unboxing comprend les étapes suivantes :

* Vérification de l'instance de l'objet pour s'assurer qu'il s'agit bien d'une valeur boxed du type valeur spécifié.
* Copie de la valeur de l'instance dans la variable de type valeur.

Les instructions suivantes expliquent les opérations de boxing et d'unboxing :

int i = 123; // a value type

object o = i; // boxing

int j = (int)o; // unboxing

Pour que l'unboxing de types valeur réussisse au moment de l'exécution, l'élément qui est unboxed doit être une référence à un objet précédemment créé par boxing d'une instance de ce type valeur. La tentative d'extraction de null provoque un NullReferenceException. La tentative d'extraction d'une référence vers un type de valeur incompatible provoque un InvalidCastException.

Les classes

Une classe définie un type d’objet mais il ne s’agit pas d’un objet.  
On peut les créer en utilisant le mot clé new suivi du nom de la classe que l’on veut créer.

Customer object1 = new Customer();

Les classes prennent en charge l’héritage. Une classe peut hériter de toute autre interface ou classe.  
Pour cela on crée une classe à partir d’une autre classe dont elle hérite des données et du comportement.  
Ici la classe manager hérite de la classe employé :

public class Manager : Employee {

}

Une classe peut être déclarée abstract. Une classe abstraite contient des méthodes abstraites qui ont une définition de signature, mais aucune implémentation. Les classes abstraites ne peuvent pas être instanciées. Elles peuvent être utilisées uniquement à travers des classes dérivées qui implémentent les méthodes abstraites.

Les interfaces (IEnumerable)

IEnumerable est l’interface de base pour toutes les collections non génériques qui peuvent être énumérées.

IEnumerable contient une méthode unique, GetEnumerator, qui retourne un [IEnumerator](https://docs.microsoft.com/fr-fr/dotnet/api/system.collections.ienumerator?view=netframework-4.8). IEnumerator offre la possibilité d’itérer au sein de la collection en exposant une propriété Current et des méthodes MoveNext et Reset.

Il est recommandé d’implémenter IEnumerable et IEnumerator sur vos classes de collection pour activer la syntaxe foreach mais l’implémentation de IEnumerable n’est pas obligatoire. Si votre collection n’implémente pas IEnumerable, vous devez toujours suivre le modèle d’itérateur pour prendre en charge cette syntaxe en fournissant une méthode GetEnumerator qui retourne une interface, une classe ou un struct.

Les Types Génériques

Les génériques sont introduits dans le .NET Framework 2.0, ils permettent de définir des structures de données de type sécurisé sans se limiter à un type de données réel.

Par exemple List<T> est une collection générique, elle peut être déclarée et utilisée avec n’importe quel type : List<int>, List<string> etc.

Les génériques permettent un gain de performances car il n’y a pas de boxing et unboxing donc on gagne du temps. De plus, avant l’implémentation des génériques, il n’y avait aucun moyen de savoir ce que le type de données dans la liste alors que ce problème est fixé car le type est prédéfini par exemple il y aura uniquement des entiers dans List<int>

//generic list

List<int> ListGeneric = new List<int> { 5, 9, 1, 4 };

//non-generic list

ArrayList ListNonGeneric = new ArrayList { 5, 9, 1, 4 };

Les génériques sont plus fréquemment utilisés dans des collections et des méthodes qui agissent sur eux

Un autre avantage de la connaissance du type de votre générique par le runtime est une meilleure expérience de débogage. Quand vous déboguez un générique en C#, vous connaissez le type de chaque élément dans votre structure de données. Sans les génériques, vous ne pourriez pas connaître le type de chaque élément.

Les délégués

Un délégué est un type qui représente des références aux méthodes. Lorsque vous instanciez un délégué, vous pouvez associer son instance à toute méthode ayant une signature et un type de retour compatibles. Les délégués sont utilisés pour passer des méthodes comme arguments à d’autres méthodes. L'exemple suivant illustre une déclaration de délégué :

public delegate int PerformCalculation(int x, int y);

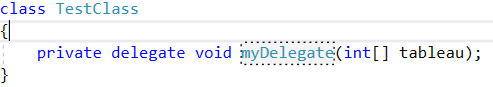
Toute méthode de n’importe quelle classe ou structure accessible qui correspond au type de délégué, peut être assignée au délégué.

Comparé à la surcharge des méthodes, les délégués incluent dans la signature la valeur de retour, il faut donc que la méthode à le même type de retour que le délégué.

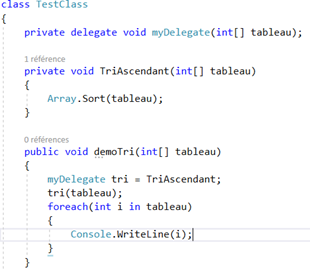
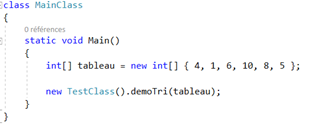
L’un des intérêts majeurs des délégués est l’appelle à plusieurs méthode (ou multidiffusion). Ce concept repose sur la liste d’invocation, elle représente la liste des méthodes du delegate. Pour créer une liste d’invocation, il suffit de faire la somme des fonctions que l’on veut utiliser avec l’opérateur « + ». L’ajout d’une nouvelle méthode se fait par l’opérateur « += ».

Les délégués sont orientés objet et encapsulent une instance d’objet et une méthode. Ils permettent aux méthodes d’être transmises comme des paramètres.

Les méthodes anonymes et les expressions lambda sont également compilées en type de délégué.



Ici, un délégué privé est créé dans la classe *TestClass* et il permettra de pointer vers des méthodes qui ne retourne rien (*void*) et qui acceptent un tableau d’entier en paramètre.

Dans ce code, nous avons implémenté une méthode *private void TriAscendant(int[] tableau)*, qui a la même signature que notre délégué.

Dans la méthode demoTri, nous commençons par déclarer une variable du type du délégué *myDeleguate*, qui est le délégué que nous avons créé.

Puis nous faisons pointer cette variable vers la méthode *TriAscendant*.

On exécute la fonction Main() et on obtient bien le tableau trié.

Les expressions Lambda

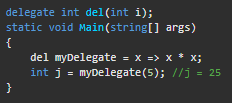
Tout d’abord une expression se caractérise par la succession de plusieurs opérandes/opérateurs ayant pour but de simplifier la déclaration de méthode, ou la surcharge d’opérateur de façon beaucoup plus concise. Les expressions lambda caractérisent une partie des expressions, celles-ci se déclarent de la façon suivante :

(input-parameters) => expression



L’opérateur => est utilisé pour séparer les paramètres du corps de l’expression. Il s’agit de fonctions locales pouvant donc prendre 0 à plusieurs paramètres (dont le types peut être spécifié) leur but étant de réaliser dans la majeure partie des cas des opérations simples.

Les expressions lambdas sont capables de créer des délégués.



Les collections de données

Les collections sont utilisées pour stocker et manipulés plus efficacement des données similaires.  
On peut utiliser la classe System.Array ou les classes présentes dans les espaces de noms System.Collection : System.Collections.Generic, System.Collections.Concurrent et System.Collections.Immutable pour ajouter, supprimer et modifier des éléments individuels ou une série d’éléments dans une collection.

Il existe les collections génériques et non-génériques. Les collections génériques fournissent des collections de type sécurisé au moment de la compilation. Pour cette raison, les collections génériques offrent généralement de meilleures performances. Les collections génériques acceptent un paramètre de type lorsqu'elles sont construites. Les collections non génériques stockent des éléments comme Object, nécessitent une conversion, et la plupart ne sont pas prises en charge pour le développement d’applications du Windows Store.

Toutes les collections fournissent des méthodes pour l'ajout, la suppression ou la recherche d'éléments au sein d'une collection.

Les collections du .NET Framework implémentent soit System.Collections.IEnumerable, soit System.Collections.Generic.IEnumerable<T> pour permettre à la collection d'être parcourue. Un énumérateur peut être vu comme un pointeur mobile pointant vers n'importe quel élément d'une collection.

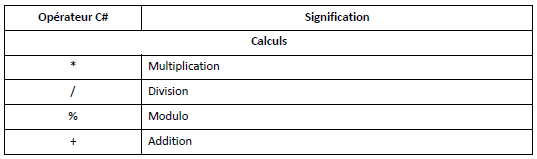
Toutes les collections peuvent être copiées dans un tableau à l'aide de la méthode CopyTo. Toutefois, l'ordre des éléments du nouveau tableau sera basé sur l'ordre où ils sont retournés par l'énumérateur. Le tableau résultant est toujours unidimensionnel avec une limite inférieure de zéro.

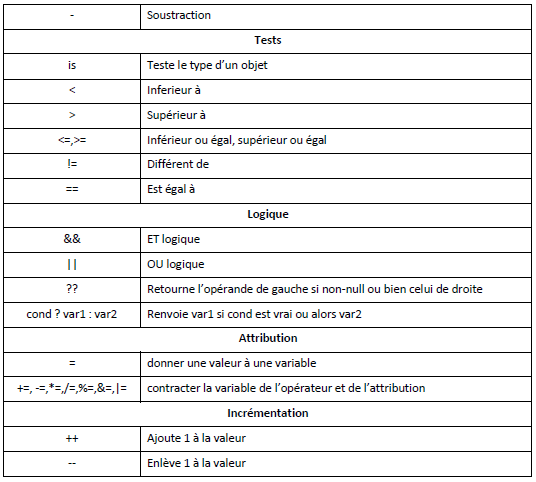
La propriété Capacity d'une collection indique le nombre d'éléments qu'elle peut contenir. La propriété Count d'une collection indique le nombre d'éléments qu'elle contient réellement. Certaines collections masquent l'une de ces propriétés, voire les deux.

La capacité de la plupart des collections s'étend automatiquement quand la valeur de la propriété Capacity est atteinte. La mémoire est réallouée et les éléments sont copiés depuis l'ancienne collection vers la nouvelle.

Les opérateurs

Les opérateurs en C# sont des fonctions spéciales fournissant les opérations primitives du langage. On en distingue 5 types : - Les opérateurs mathématiques - Les opérateurs booléens - Les opérateurs logiques - Les opérateurs d’attribution - Les opérateurs d’incrémentation





Les indexeurs

Les indexeurs

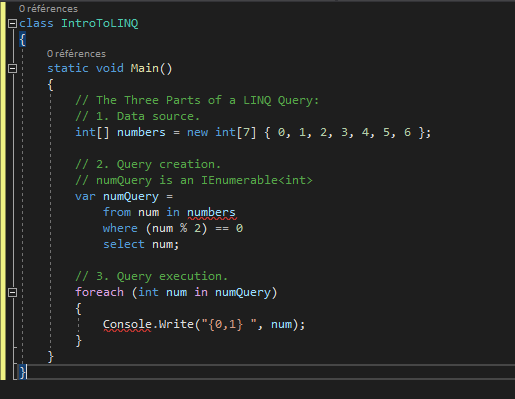
LINQ

LINQ est l’acronyme de « Language-Integrated Query » permet d’effectuer des requêtes à des collections d’objet stockés en mémoire plus simplement. Pour effectuer une requête LINQ, il faut :

Récupérer la source de données (une liste d’entier par exemple ou n’importe quelle collection d’objets)

Créer une requête en utilisant les mots clés de l’API Standard Query Operator (Select, Where, …)

Et enfin, exécuter la requête.



Dans l’exemple précédent on voit que la requête est du type IEnumerable. Le type IEnumerable ainsi que toutes les interfaces qui dérivent de cette classe sont considérées comme des type requêtable. Une variable ayant un type requêtable est une source de données LINQ de base. Cependant il existe aussi des sources de données qui n’ont pas de type considéré comme requêtable, l’API qui gère LINQ de convertir ces types pour qu’ils deviennent requêtable.

Si les données sources ne sont pas déjà en mémoire en tant que type interrogeable, le fournisseur LINQ doit le représenter comme tel. Par exemple, LINQ to XML charge un document XML dans un type [XElement](https://docs.microsoft.com/fr-fr/dotnet/api/system.xml.linq.xelement) Requêtable. LINQ permet aussi de créer des requêtes pour des bases de données SQL classique. Pour cela, il suffit de donner le type IQueryable (qui dérive de IEnumable) à la requête LINQ.

La requête

La requête spécifie les informations à récupérer à partir de la ou des sources de données. Si vous le souhaitez, une requête peut également spécifier la manière dont ces informations doivent être triées, regroupées et mises en forme avant d’être retournées. Une requête est stockée dans une variable de requête et initialisée avec une expression de requête. Pour faciliter l’écriture de requêtes, le langage C# propose désormais une nouvelle syntaxe de requête.

Comme pour le SQL, il y a plusieurs mots clés :

Gestion des exceptions

Les attributs et réflexion