Labo.net SUPINFO

http://www.labo-dotnet.com

Programmation en C#

SUPINFO DOT NET TRAINING COURSE

Auteur : Ary Quloré et Mathieu Szablowski

Version 2.0 – 10 novembre 2004
Nombre de pages : 83

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Table des matières

1. Vue d'ensemble de la plate-forme Microsoft .NET

1.1. Présentation de la plate-forme .NET

La plate-forme Microsoft® .NET fournit l'ensemble des outils et technologies nécessaires à la création d'applications Web distribuées. Elle expose un modèle de programmation cohérent, indépendant du langage, à tous les niveaux d'une application, tout en garantissant une parfaite interopérabilité avec les technologies existantes et une migration facile depuis ces mêmes technologies.

La plate-forme .NET prend totalement en charge les technologies Internet basées sur les normes et indépendantes des plates-formes, telles que http (*Hypertext Transfer Protocol*), XML (*Extensible Markup Language*) et SOAP (*Simple Object Access Protocol*).

C# est un nouveau langage spécialement conçu pour la création d'applications .NET.
En tant que développeur, il vous sera utile de comprendre le fondement et les fonctionnalités de la plate-forme Microsoft .NET avant d'écrire du code C#.

La plate-forme .NET offre plusieurs technologies de base. Ces technologies sont décrites dans les rubriques suivantes.

1.1.1. .NET Framework

La technologie du .NET Framework se fonde sur un nouveau Common Language Runtime. Celui-ci fournit un ensemble commun de services pour les projets créés dans Microsoft Visual Studio® .NET, indépendamment du langage. Ces services fournissent des blocs de construction de base pour les applications de tous types, utilisables à tous les niveaux des applications.
Microsoft Visual Basic®, Microsoft Visual C++® et d'autres langages de programmation de Microsoft ont été améliorés pour tirer profit de ces services.
Microsoft Visual J#™ .NET a été créé pour les développeurs Java qui souhaitent créer des applications et des services à l'aide du .NET Framework. Les langages tiers écrits pour la plate-forme .NET ont également accès à ces services. Le .NET Framework est présenté plus en détails dans la suite de ce module.

1.1.2. .NET My Services

.NET My Services est un ensemble de services Web XML centrés sur l'utilisateur. Grâce à .NET My Services, les utilisateurs reçoivent des informations pertinentes lorsqu'ils en ont besoin, directement sur les périphériques qu'ils utilisent en fonction des préférences qu'ils ont définies.
Grâce à .NET My Services, les applications peuvent communiquer directement via SOAP et XML à partir de toute plate-forme prenant en charge SOAP.

Les principaux services .NET My Services sont les suivants :
- Authentification .NET Passport
- Possibilité d'envoyer des alertes et de gérer les préférences pour la réception des alertes
- Stockage d'informations personnelles (contacts, adresses électroniques, calendriers, profils, listes, portefeuille électronique et emplacement physique)
- Possibilité de gérer des banques de documents, d'enregistrer les paramètres des applications, de sauvegarder vos sites Web préférés et de noter les périphériques possédés

1.1.3. Serveurs .NET Entreprise Server

Les serveurs .NET Enterprise Server permettent une évolutivité, une fiabilité, une gestion et une intégration inter et intra organisations. Ils offrent en outre un grand nombre de fonctionnalités décrites dans le tableau ci-dessous.

1.1.4. Visual Studio .NET

Visual Studio .NET constitue un environnement de développement destiné à la création d'applications sur le .NET Framework. Il fournit d'importantes technologies catalysantes afin de simplifier la création, le déploiement et l'évolution continue d'applications Web et de services Web XML sécurisés, évolutifs et à haute disponibilité.

1.2. Vue d'ensemble du Framework .NET

1.2.1. .NET Framework

.NET Framework fournit le canevas de compilation et d'exécution nécessaire à la création et à l'exécution d'applications .NET.

1.2.2. Substrat de la plate-forme

Le .NET Framework doit être exécuté sur un système d'exploitation.
Actuellement, le .NET Framework est conçu pour fonctionner sur les systèmes d'exploitation Microsoft Win32®. Dans l'avenir, il sera étendu pour s'exécuter sur d'autres plates-formes, telles que Microsoft Windows® CE.

1.2.3. Services d'application

En exécutant le .NET Framework sur Windows, les développeurs disposent de certains services d'application, tels que Component Services, Message Queuing, Windows Internet Information Services (IIS) et Windows Management Instrumentation (WMI). Le .NET Framework expose ces services d'application par l'intermédiaire de classes de sa bibliothèque de classes.

1.2.4. Common Laguage Runtime

Le Common Language Runtime facilite le développement d'applications, fournit un environnement d'exécution robuste et sécurisé, prend en charge plusieurs langages de programmation tout en simplifiant le déploiement et la gestion des applications.
Son environnement est également qualifié de « managé », ce qui signifie que des services courants, tels que le garbage collection et la sécurité, y sont automatiquement fournis.

1.2.5. Bibliothèque de classe

La bibliothèque de classes .NET Framework expose des fonctionnalités Runtime et fournit d'autres services utiles à tous les développeurs. Les classes simplifient le développement des applications .NET. Les développeurs peuvent ajouter des classes en créant leurs propres bibliothèques de classes.

1.2.6. ADO.NET

ADO.NET est la nouvelle génération de la technologie ADO (*ActiveX® Data Object*) de Microsoft. Elle fournit une prise en charge améliorée du modèle de programmation déconnecté, ainsi qu'une prise en charge riche du XML.

1.2.7. ASP.NET

Microsoft ASP.NET est une structure de programmation fondée sur le Common Language Runtime. Il peut être employé sur un serveur pour créer des applications Web puissantes. Les formulaires Web ASP.NET sont des outils puissants et faciles d'emploi permettant de créer des interfaces utilisateur Web dynamiques.

1.2.8. Services Web XML

Un service Web est un composant Web programmable qui peut être partagé par des applications sur Internet ou sur un intranet. Le .NET Framework fournit des outils et des classes pour la création, le test et la distribution de services Web XML.

1.2.9. Interfaces utilisateur

Le .NET Framework prend en charge trois types d'interfaces utilisateur :

- Les formulaires Web, qui fonctionnent avec ASP.NET.
- Les Windows Forms, qui fonctionnent sur les clients Win32.
- Les applications consoles, qui, par souci de simplicité, sont utilisées pour la plupart des ateliers de ce cours.

1.2.10. Langages

Tout langage conforme à la norme CLS (Common Language Specification) peut fonctionner dans le Common Language Runtime. Dans le .NET Framework, Microsoft prend en charge Visual Basic, Visual C++, Microsoft Visual C#, Visual J# et Microsoft JScript®. Les fournisseurs indépendants peuvent proposer d'autres langages.

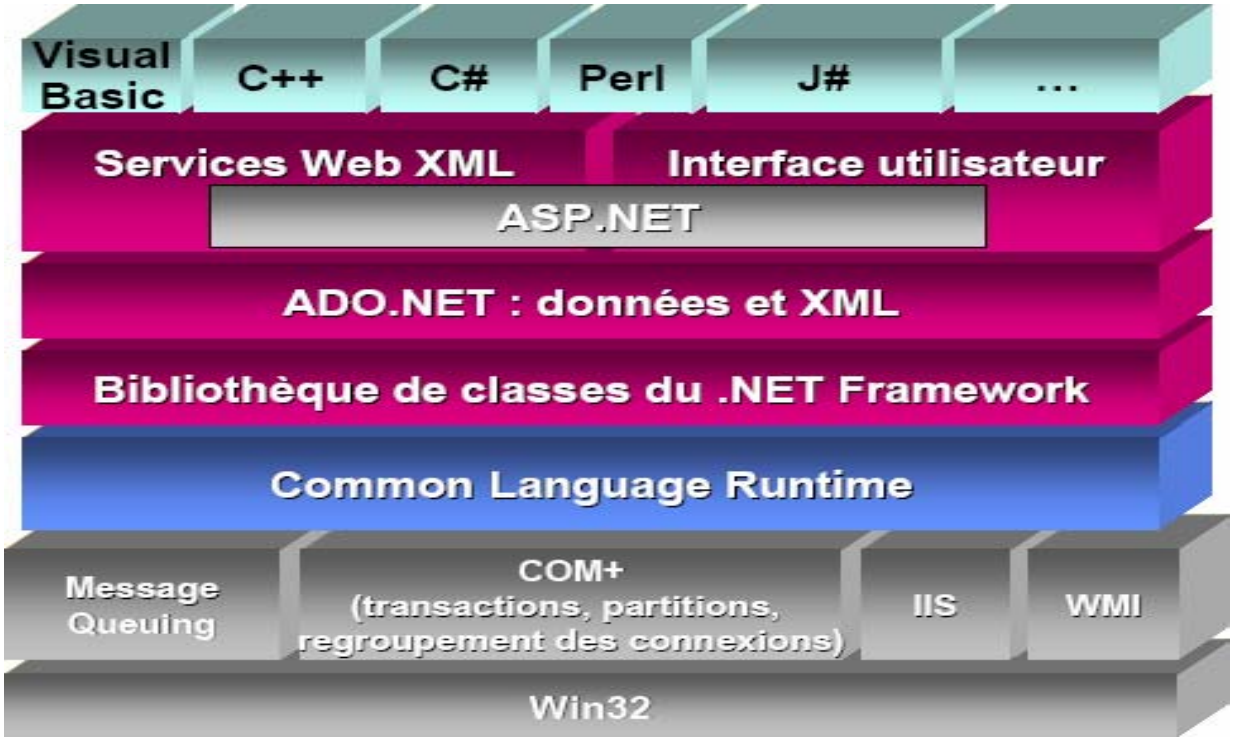

Figure 1 - Plateforme .NET

1.3. Avantages Framework .NET

Dans cette section, vous allez découvrir certains des avantages du .NET Framework.
Le .NET Framework a été conçu pour répondre aux objectifs suivants :

1.3.1. S'appuyer sur les normes et les pratiques du Web

Il prend totalement en charge les technologies Internet existantes, telles que HTML, XML, SOAP, XSLT (*Extensible Stylesheet Language for Transformations*), XPath et autres normes Web. Il favorise les services Web XML sans état et à connexion non permanentes.

1.3.2. Utiliser des modèles d'application unifiés

La fonctionnalité d'une classe .NET est accessible à partir de tous les modèles de programmation ou langages compatibles avec .NET.

1.3.3. Facile d'emploi pour les développeurs

Dans le .NET Framework, le code est organisé en classes et en espaces de noms hiérarchiques. Le .NET Framework fournit un système de types communs, également qualifié de « système de types unifiés », qui peut être utilisé par tous les langages compatibles avec .NET. Dans ce système de types unifiés, tous les éléments du langage sont des objets. Il n'existe aucun type ***Variant***, uniquement un type ***String***, et toutes les données de type ***String*** sont au format ***Unicode***. Le système unifié est décrit plus en détail dans les modules ultérieurs.

1.3.4. Classes extensibles

La hiérarchie du .NET Framework est visible par le développeur. Vous pouvez accéder aux classes .NET et les étendre (à l'exception des classes protégées) au moyen de l'héritage. Il est également possible d'implémenter un héritage inter langage.

1.4. Composants du Framework .NET

Dans cette section, vous allez vous familiariser avec Microsoft .NET Framework. Le .NET Framework est un ensemble de technologies intégrées à la plate-forme Microsoft .NET. Cet ensemble constitue les blocs de construction de base qui servent au développement d'applications et de services Web XML.

1.4.1. Common Language Runtime

Le Common Language Runtime facilite le développement d'applications, fournit un environnement d'exécution robuste et sécurisé, prend en charge plusieurs langages de programmation tout en simplifiant le déploiement et la gestion des applications. Le Runtime est appelé *environnement managé* et fournit automatiquement des services communs tels que le garbage collection, la sécurité, etc. Les fonctionnalités du Common Language Runtime sont décrites dans le tableau ci-dessous.

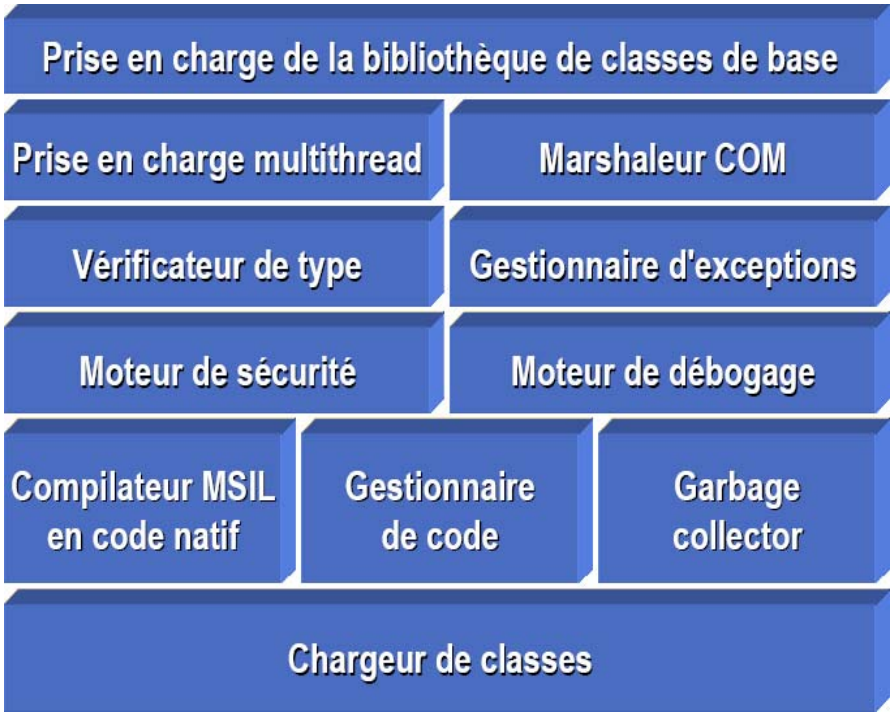

Figure 2 - Fonctionnalités de la CLR

Composant	Description
Chargeur de classes	Gère les métadonnées, en plus du chargement des classes et de leur disposition.
Compilateur MSIL (*Microsoft Intermediate Language*) en code natif	Convertit MSIL en code natif (juste-à-temps).
Gestionnaire de code	Gère l'exécution du code.
Garbage collector (GC)	Fournit une gestion automatique de la durée de vie de tous vos objets. Il s'agit d'un mécanisme multiprocesseur et évolutif.
Moteur de sécurité	Fournit une sécurité par preuve, fondée sur l'origine du code en plus de l'utilisateur.
Moteur de débogage	Permet de déboguer l'application et de tracer l'exécution du code.
Vérificateur de type	N'autorise pas les conversions non sécurisées ou les variables non initialisées. Le langage MSIL peut être vérifié pour garantir la sécurité de type.
Gestionnaire d'exceptions	Fournit un traitement structuré des exceptions, intégré à SEH (*WindowsStructured Exception Handling*). La génération de rapports d'erreurs a été améliorée.
Prise en charge multithread	Fournit des classes et des interfaces qui permettent la programmation multithread.
Marshaleur COM	Fournit le marshaling à partir et à destination de COM.
Prise en charge de la bibliothèque de classes de base (BCL)	Intègre du code au runtime qui prend en charge la BCL.

Tableau 1 - Description des composants

1.4.2. Bibliothèque de classes du .NET Framework

La bibliothèque de classes du .NET Framework expose les fonctionnalités du Runtime et fournit d'autres services essentiels de haut niveau via une hiérarchie d'objets.

- ***Espace de noms System***

L'espace de noms ***System*** contient des classes fondamentales et de base qui définit les types de données valeur et référence, les événements et gestionnaires d'événements, les interfaces, les attributs et les exceptions de traitements couramment utilisés. D'autres classes fournissent des services qui prennent en charge les conversions des types de données, la manipulation de paramètres de méthode, les opérations mathématiques, l'appel de programmes à distance et en local, la gestion d'environnements d'applications, de même que la supervision d'applications gérées et non gérées.

L'espace de noms ***System.Collections*** fournit des listes triées, des tables de hachage et d'autres méthodes de regroupement de données. L'espace de noms ***System.IO*** fournit des entrées/sorties (E/S), des flux, etc. L'espace de noms ***System.NET*** offre une prise en charge des sockets et de TCP/IP (*Transmission Control Protocol/Internet Protocol*).

1.4.3. ADO.NET : données et XML

ADO.NET, la nouvelle génération de la technologie ADO, fournit une prise en charge améliorée du modèle de programmation déconnecté, ainsi qu'une prise en charge riche du XML dans l'espace de noms ***System.Xml***.

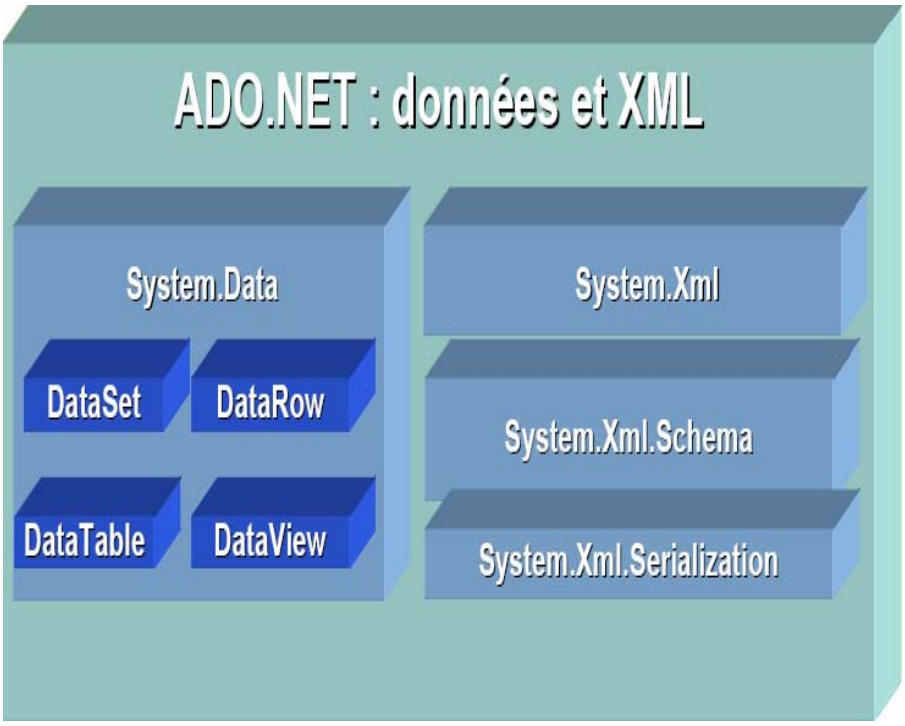

Figure 3 - Espaces de noms composant ADO.NET

- ***Espace de noms System.Data***

L'espace de noms ***System.Data*** est constitué de classes qui forment le modèle objet d'ADO.NET. À haut niveau, le modèle objet ADO.NET se divise en deux couches : la couche connectée et la couche déconnectée.
L'espace de noms ***System.Data*** comprend la classe ***DataSet***, qui représente des plusieurs tables et leurs relations. Ces ensembles ***DataSets*** sont des structures de données totalement autonomes, qui peuvent être peuplées à partir de diverses sources de données. L'une de ces sources pourrait être du code XML, une autre, une base de données OLE et une troisième, l'adaptateur direct pour SQL Server.

- ***Espace de noms System.Xml***

L'espace de noms ***System.Xml*** fournit la prise en charge XML. Il comprend un outil d'écriture et un analyseur XML, tous deux conformes aux spécifications du W3C. La transformation XSL (*Extensible Stylesheet Language*) est assurée par l'espace de noms ***System.Xml.Xsl***. L'espace de noms ***System.Xml.Serialization*** contient des classes utilisées pour la sérialisation des objets dans des documents ou des flux au format XML.

1.4.4. Formulaires Web et services Web XML

ASP.NET est une structure de programmation fondée sur le Common Language Runtime, qui peut être employée sur un serveur pour créer des applications Web puissantes. Les formulaires Web ASP.NET sont des outils puissants et faciles d'emploi permettant de créer des interfaces utilisateur Web dynamiques.
Les services Web ASP.NET fournissent les blocs de construction servant à l'élaboration d'applications Web XML distribuées. Les services Web XML reposent sur des standards Internet ouverts, tels que HTTP et XML.

Le Common Language Runtime fournit un support intégré pour la création et l'exposition de services Web XML, grâce à l'emploi d'une abstraction de programmation cohérente et familière à la fois pour les développeurs de formulaires Web ASP (*Active Server Pages*) et Visual Basic. Le modèle résultant est à la fois évolutif et extensible. Ce modèle reposant sur des standards Internet ouverts (HTTP, XML, SOAP et SDL), tout ordinateur client ou périphérique Internet peut y accéder et l'interpréter.

- ***Espace de noms System.Web***

Dans l'espace de noms ***System.Web***, certains services de niveau inférieur, tels que la mise en cache, la sécurité ou la configuration, sont partagés par les services Web XML et l'interface utilisateur Web.

- ***Espace de noms System.Web.Services***

L'espace de noms ***System.Web*** fournit des classes et des interfaces permettant la communication entre les navigateurs et les serveurs.

- ***Espace de noms System.Web.UI***

L'espace de noms ***System.Web.UI*** fournit des classes et des interfaces qui vous permettent de créer des contrôles et des pages s'affichant dans vos applications Web comme interface utilisateur sur une page Web.

1.4.5. Interface utilisateur pour Windows

- ***Espace de noms System.Windows.Forms***

Vous pouvez utiliser les classes de l'espace de noms ***System.Windows.Forms*** pour créer l'interface utilisateur cliente. Cette classe vous permet d'implémenter l'interface utilisateur Windows standard dans vos applications .NET.
De nombreuses fonctions qui n'étaient auparavant accessibles que par l'intermédiaire d'appels d'api sont à présent intégrées aux formulaires eux-mêmes, ce qui rend le développement plus puissant et plus facile.

- ***Espace de noms System.Drawing***

L'espace de noms ***System.Drawing*** fournit un accès aux fonctionnalités graphiques de base de GDI+. Des fonctionnalités plus avancées sont fournies dans les espaces de noms :
- System.Drawing.Drawing2D
- System.Drawing.Imaging
- System.Drawing.Text.

1.5. Langages du .NET Framework

Le .NET Framework prend en charge plusieurs langages de programmation. C# est le langage de programmation spécialement conçu pour la plate-forme .NET, mais C++ et Visual Basic ont également été mis à jour pour prendre entièrement en charge le .NET Framework.

Langage	Description
C#	C# a été conçu pour la plate-forme .NET. Il s'agit du premier langage moderne orienté composant dans la famille C et C++. Il peut être incorporé dans des pages ASP.NET. Certaines de ses principales fonctionnalités sont : les classes, les interfaces, les délégués, le boxing et l'unboxing, les espaces de noms, les propriétés, les indexeurs, les événements, la surcharge d'opérateurs, la gestion de versions, les attributs, le code non sécurisé et la génération de documents XML. Aucun en-tête ou fichier IDL (*Interface Definition Language*) n'est nécessaire.
Extensions managées pour C++	Le langage C++ managé est une extension minimale du langage C++. Cette extension fournit un accès au .NET Framework qui comprend le garbage collection, l'héritage d'implémentation simple et l'héritage d'interface multiple. Cette mise à jour dispense également le développeur d'écrire du code de raccord pour les composants. Il offre un accès de bas niveau si besoin est.
Visual Basic .NET	Visual Basic .NET présente diverses innovations importantes par rapport aux versions précédentes de Visual Basic. Visual Basic .NET prend en charge l'héritage, les constructeurs, le polymorphisme, la surcharge du constructeur, les exceptions structurées, un contrôle de type plus strict, des modèles de thread libres ainsi que de nombreuses autres fonctionnalités. Il existe une seule forme d'assignation : aucune méthode ***Let*** ou ***Set***. Les nouvelles fonctions de développement rapide d'application, telles que le Concepteur XML, l'Explorateur de serveurs et le Concepteur Web Forms, sont disponibles dans Visual Basic à partir de Visual Studio .NET. Avec cette version, VBScript offre une fonctionnalité Visual Basic complète.
JScript .NET	JScript .NET est réécrit pour prendre entièrement en charge .NET. Il prend en charge les classes, l'héritage, les types et la compilation, et comprend des fonctionnalités de performance et de productivité optimisées. JScript .NET est également intégré avec Visual Studio .NET. Vous pouvez utiliser n'importe quelle classe .NET Framework dans JScript .NET.
Visual J# .NET	Visual J# .NET est un outil de développement Java destiné aux développeurs Java qui souhaitent créer des applications et des services sur le .NET Framework. Visual J# .NET est entièrement compatible avec .NET et inclut des outils pour mettre à jour et convertir automatiquement les projets et les solutions Visual J++ 6.0 existants au nouveau format Visual Studio .NET. Visual J# .NET fait partie de la stratégie de transition de Java à .NET (*Java User Migration Path to Microsoft .NET, « JUMP to .NET »*).
Langages tiers	Divers langages tiers prennent en charge le .NET Framework. Citons, par exemple, APL, COBOL, Pascal, Eiffel, Haskell, ML, Oberon, Perl, Python, Scheme et SmallTalk.

Tableau 2 - Description des langages

2. Vue d'ensemble du C#

2.1. Structure du programme C#

Dans cette leçon, vous allez apprendre la structure de base d'un programme C#.
Vous allez analyser un programme simple contenant toutes les fonctionnalités essentielles. Vous apprendrez également à utiliser Microsoft® Visual Studio® pour créer et modifier un programme C#.

2.1.1. Hello World

Le premier programme rédigé par la plupart des utilisateurs dans un nouveau langage est l'inévitable « ***Hello, World*** ». Dans ce module, vous allez pouvoir examiner la version C# de ce premier programme traditionnel.

L'exemple de code de la diapositive contient tous les éléments essentiels d'un programme C#, et il est facile à tester ! Lorsqu'il est exécuté depuis la ligne de commandes, il affiche simplement :

Hello, World

Dans les rubriques suivantes, vous allez analyser ce programme simple pour découvrir plus en détail les blocs de construction d'un programme C#.

```
public class Hello
{
	public static void Main()
	{
		System.Console.WriteLine("Hello, World!");
	}
}
```

2.1.2. La classe

Dans C#, une application est une collection d'une ou plusieurs classes, structures de données et autres types. Dans ce module, une classe est définie comme un ensemble de données associé à des méthodes (ou fonctions) qui peuvent manipuler ces données. Dans les modules suivants, vous étudierez les classes et tout ce qu'elles offrent aux programmeurs en C#.

Si vous étudiez le code de l'application « ***Hello, World*** », vous verrez une seule classe appelée ***Hello***. Cette classe est introduite par le mot-clé ***class***. Le nom de la classe est suivi d'une accolade ouvrante (***{***). Tous les éléments compris entre cette accolade ouvrante et l'accolade fermée (***}***) font partie de la classe.

Vous pouvez répartir les classes d'une application C# dans un ou plusieurs fichiers. Vous pouvez placer plusieurs classes dans un fichier, mais vous ne pouvez pas étendre une même classe sur plusieurs fichiers.

```
public class Hello
{
	...
}
```

2.1.3. La méthode Main

Chaque application doit démarrer à un emplacement donné. L'exécution d'une application C# démarre au niveau de la méthode ***Main***. Si vous avez l'habitude de programmer en langage C, C++ ou Java, vous êtes déjà familiarisé avec ce concept.

Le langage C# respecte la casse. ***Main*** doit être écrit avec un « ***M*** » majuscule et le reste du mot en minuscules.

Une application C# peut contenir plusieurs classes, mais un seul point d'entrée. Vous pouvez avoir plusieurs classes avec ***Main*** dans la même application, mais une seule méthode ***Main*** est exécutée. Vous devez spécifier la classe à utiliser lors de la compilation de l'application.

La signature de la méthode ***Main*** est également importante. Si vous utilisez Visual Studio, cette signature est créée automatiquement en tant que ***static void***. (Vous découvrirez ce que cela signifie plus loin dans ce cours.) À moins d'avoir une bonne raison de le faire, vous ne devez pas modifier la signature.

Vous pouvez modifier la signature dans une certaine mesure, mais elle doit toujours être statique ; sinon, elle peut ne pas être reconnue par le compilateur comme point d'entrée de l'application.

L'application s'exécute jusqu'à la fin de la méthode ***Main*** ou jusqu'à ce qu'une instruction ***return*** soit exécutée par la méthode ***Main***.

2.1.4. La directive using et l'espace de noms System

Dans Microsoft .NET Framework, C# est fourni avec de nombreuses classes d'utilitaires qui effectuent diverses opérations utiles. Ces classes sont organisées en ***espaces de noms***. Un espace de noms est un ensemble de classes associées. Il peut également contenir d'autres espaces de noms.

Le .NET Framework se compose de nombreux espaces de noms, le plus important étant ***System***. L'espace de noms ***System*** contient les classes que la plupart des applications utilisent pour interagir avec le système d'exploitation. Les classes les plus fréquemment utilisées gèrent les entrées/sorties (E/S). Comme de nombreux autres langages, C# n'a pas de capacité d'E/S à part entière ; il dépend par conséquent du système d'exploitation pour fournir une interface compatible avec C#.

Vous pouvez faire référence à des objets d'un espace de noms en les faisant précéder explicitement de l'identificateur de cet espace de noms. Par exemple, l'espace de noms ***System*** contient la classe ***Console***, qui comporte plusieurs méthodes, telles que ***WriteLine***. Vous pouvez accéder à la méthode ***WriteLine*** de la classe ***Console*** de la manière suivante :

```
System.Console.WriteLine("Hello, World!");
```

Toutefois, l'utilisation d'un nom qualifié complet pour faire référence aux objets est dangereuse et risque de provoquer des erreurs. Afin d'éviter ce problème, vous pouvez spécifier un espace de noms en plaçant une directive ***using*** au début de votre application avant la définition de la première classe. Une directive ***using*** spécifie un espace de noms qui sera étudié si une classe n'est pas explicitement définie dans l'application. Vous pouvez intégrer plusieurs directives ***using*** dans le fichier source, mais elles doivent toutes être placées au début de ce fichier.

Avec la directive ***using***, vous pouvez réécrire le code précédent de la manière suivante :

```
using System;

public class Hello
{
      public static void Main()
      {
            Console.WriteLine("Hello, World!");
      }
}
```

Dans l'application « ***Hello, World*** », la classe ***Console*** n'est pas explicitement définie. Lors de la compilation de l'application « ***Hello, World*** », le compilateur recherche la classe ***Console*** et la trouve dans l'espace de noms ***System***.
Le compilateur génère du code qui fait référence au nom qualifié complet ***System.Console***.

2.2. Opérations élémentaires d'entrée/sortie

Dans cette leçon, vous allez apprendre à effectuer des opérations d'entrée/sortie à partir de commandes en langage C# à l'aide de la classe ***Console***.
Vous apprendrez à afficher des informations à l'aide des méthodes ***Write*** et ***WriteLine***, ainsi qu'à recueillir des informations d'entrée à partir du clavier à l'aide des méthodes ***Read*** et ***ReadLine***.

2.2.1. La classe Console

La classe ***Console*** fournit une application C# avec accès à l'entrée standard, à la sortie standard et aux flux d'erreur standard.

L'entrée standard est normalement associée au clavier ; tout ce que l'utilisateur saisit à partir du clavier peut être lu à partir du flux d'entrée standard. Le flux de sortie standard et le flux d'erreur standard sont quant à eux généralement associés à l'écran.

- ***Méthodes Write et WhriteLine***

Vous pouvez utiliser les méthodes ***Console.Write*** et ***Console.WriteLine*** pour afficher des informations sur l'écran de la console. Ces deux méthodes sont très similaires, à une différence près : la méthode ***WriteLine*** ajoute une nouvelle paire ligne/retour chariot à la fin de la sortie, tandis que la méthode ***Write*** ne le fait pas.

Ces deux méthodes sont surchargées. Vous pouvez les appeler avec différents nombres et types de paramètres. Par exemple, vous pouvez utiliser le code suivant pour écrire « ***99*** » à l'écran :

```
Console.WriteLine(99);
```

Vous pouvez utiliser le code suivant pour écrire le message « ***Hello, World*** » à l'écran :

```
Console.WriteLine("Hello, World!");
```

- ***Méthodes Read et ReadLine***

Vous pouvez lire l'entrée de l'utilisateur sur le clavier à l'aide des méthodes ***Console.Read*** et ***Console.ReadLine***.

La méthode ***Read*** lit le caractère suivant à partir du clavier. Elle renvoie la valeur ***int –1*** si aucune autre entrée n'est disponible. Sinon, elle renvoie une valeur ***int*** représentant le caractère lu.

La méthode ***ReadLine*** lit tous les caractères jusqu'à la fin de la ligne d'entrée (le retour chariot). L'entrée est renvoyée en tant que chaîne de caractères. Vous pouvez utiliser le code ci-dessous pour lire une ligne de texte à partir du clavier et l'afficher à l'écran :

```
string input = Console.ReadLine();
Console.WriteLine(input);
```

2.3. Méthodes conseillées

Dans cette leçon, vous allez apprendre certaines méthodes conseillées lors de l'écriture d'applications C#. Vous découvrirez comment commenter les applications afin d'optimiser leur lecture et leur gestion. Vous apprendrez également à gérer les erreurs qui peuvent se produire lors de l'exécution d'une application.

2.3.1. Commentaire d'applications

Il est important de fournir une documentation adéquate pour toutes vos applications. Vous devez fournir des commentaires suffisants pour qu'un développeur n'ayant pas participé à la création de l'application d'origine soit en mesure de suivre et de comprendre son fonctionnement. Utilisez des commentaires précis et explicites. Les bons commentaires ajoutent des informations que l'instruction de code seule ne suffit pas à expliciter ; ils expliquent le « pourquoi » plutôt que le « qu'est-ce que c'est ». Si votre entreprise applique des règles en matière de commentaires, vous devez les respecter.

C# permet d'ajouter des commentaires au code des applications de plusieurs manières : commentaires sur une seule ligne, commentaires sur plusieurs lignes ou documentation générée au format XML.

Vous pouvez ajouter un commentaire sur une seule ligne en utilisant les barres obliques (`//`). Lors de l'exécution de votre application, tout ce qui suit ces deux caractères jusqu'à la fin de la ligne est ignoré.

```
// lit le nom de l'utilisateur
Console.WriteLine("Comment vous appelez-vous?");
name = Console.ReadLine();
```

Vous pouvez également utiliser des commentaires de bloc qui couvrent plusieurs lignes. Ce type de commentaires commence par la paire de caractères
`/* et continue jusqu'à la paire de caractères */` correspondante. Vous ne pouvez pas imbriquer de commentaires de bloc.

```
/* Recherche la racine la plus élevées de
L'équation quadratique */
x = (...);
```

2.3.2. Création d'une documentation XML

Vous pouvez utiliser des commentaires C# pour générer une documentation XML pour vos applications.

Les commentaires des documentations commencent par trois barres obliques (///) suivies d'une balise de documentation XML. Pour consulter des exemples, reportez-vous à la diapositive.

Vous pouvez utiliser les balises XML qui vous sont proposées, ou créer les vôtres. Le tableau suivant contient certaines balises XML et leur utilisation.

Balise	Rôle
<summary>…</summary>	Fournir une brève description. Utilisez la balise <remarks> pour une description plus longue.
<remarks> … </remarks>	Fournir une description détaillée. Cette balise peut contenir des paragraphes imbriqués, des listes et d'autres types de balises.
<para> … </para>	Ajouter une structure à la description dans une balise <remarks>. Cette balise permet de délimiter des paragraphes.
<list type="…"> … </list>	Ajouter une liste structurée à une description détaillée. Les types de listes pris en charge sont « bullet » (à puce), « number » (numérotée) et « table » (tableau). Des balises supplémentaires (<term> … </term> et <description> … </description>) sont utilisées au sein de la liste pour mieux définir la structure.
<example> … </example>	Fournir un exemple de l'utilisation d'une méthode, d'une propriété ou d'un autre membre de la bibliothèque. Cela implique souvent l'utilisation d'une balise <code> imbriquée.
<code> … </code>	Indiquer que le texte joint est un code d'application.
<c> … </c>	Indiquer que le texte joint est un code d'application. La balise <code> est utilisée pour les lignes de code qui doivent être séparées de toute description jointe ; la balise <c> est utilisée pour le code incorporé à une description jointe.
<see cref="*member*"/>	Indiquer la référence à un autre membre ou champ. Le compilateur vérifie que ce « membre » existe réellement.
<seealso cref="*member*"/>	Indiquer la référence à un autre membre ou champ. Le compilateur vérifie que ce « membre » existe réellement. La différence entre les balises <see> et <seealso> dépend du processeur qui manipule le document XML une fois ce dernier généré. Le processeur doit pouvoir générer les sections See (Voir) et See Also (Voir aussi) pour que ces deux balises soient correctement différenciées.
<exception> … </exception>	Fournir une description pour une classe d'exception.
<permission> … </permission>	Documenter l'accessibilité d'un membre.
<param name="*name*"> … </param>	Fournir une description pour un paramètre de méthode.
<returns> … </returns>	Documenter la valeur renvoyée et le type de méthode.
<value> … </value>	Décrire une propriété.

Tableau 3 - Balises XML de la documentation

Vous pouvez compiler les balises XML et la documentation dans un fichier XML en utilisant le compilateur C# avec l'option /doc :

```
csc myprogram.cs /doc:mycomments.xml
```

S'il n'y a pas d'erreur, vous pouvez visualiser le fichier XML généré à l'aide d'un outil tel qu'Internet Explorer.

Le rôle de l'option /doc consiste uniquement à générer un fichier XML. Pour afficher le fichier, vous avez besoin d'autre processeur. L'affichage du fichier sous Internet Explorer est simple, se limitant à rendre sa structure ; il permet le développement ou la réduction des balises, mais il n'affiche pas la balise ***<list type="bullet">*** en tant que puce.

- ***Exemple :***

```
/// <summary> La classe Hello affiche un message
/// de bienvenue à l'écran
/// </summary>
public class Hello
{
	/// <remarks> On utilisons l'E/S de la console.
	/// Pour plus d'informations sur WriteLine, consultez
	/// <seealso cref="System.Console.WriteLine"/>
	/// </remarks>
	public static void Main()
	{
		Console.WriteLine("Hello, World!");
	}
}
```

2.3.3. Gestion des exception

Une application C# solide doit pouvoir gérer l'inattendu. Quel que soit le système de détection d'erreurs ajouté à votre code, vous n'êtes pas à l'abri d'un problème.
Il se peut que l'utilisateur entre une réponse inattendue à une invite ou tente d'enregistrer un fichier dans un dossier supprimé. Les possibilités sont infinies.

Si une erreur d'exécution se produit dans une application C#, le système d'exploitation renvoie une exception. Interceptez les exceptions à l'aide d'une construction ***try-catch***, comme illustré sur la diapositive. Si l'une des instructions de la partie ***try*** de l'application provoque une exception, l'exécution est transférée au bloc ***catch***.

Vous pouvez rechercher des informations sur l'exception à l'aide des propriétés ***StackTrace***, ***Message*** et ***Source*** de l'objet ***Exception***. Vous en apprendrez davantage sur la gestion des exceptions dans un module ultérieur.

Si vous imprimez une exception, à l'aide de la méthode ***Console.WriteLine*** par exemple, l'exception est automatiquement formatée et affiche les propriétés ***StackTrace***, ***Message*** et ***Source***.

```
using System;

public class Hello
{
	public static void Main(string[] args)
	{
		try {
			Console.WriteLine("Hello, World!");
		}
		catch (Exception e) {
			Console.WriteLine("Exception à !{0}", e.StackTrace);
		}
	}
}
```

2.4. Compilation, exécution et débogage

Dans cette leçon, vous allez apprendre à compiler et déboguer les programmes C#. Vous découvrirez l'exécution du compilateur à partir de la ligne de commande et depuis l'environnement Visual Studio. Vous apprendrez les principales options du compilateur. Le débogueur Visual Studio vous sera présenté. Enfin, vous apprendrez à utiliser une partie des autres outils fournis avec le Kit de développement Microsoft .NET Framework SDK.

2.4.1. Appel au compilateur

Avant d'exécuter une application C#, vous devez la compiler. Le compilateur convertit le code source que vous écrivez en un code machine compris par l'ordinateur. Vous pouvez appeler le compilateur C# à partir de la ligne de commande ou depuis Visual Studio.

Plus précisément, les applications C# sont compilées en langage MSIL (Microsoft Intermediate Language) plutôt qu'en code machine natif.
Le code MSIL est lui-même compilé en code machine par le compilateur juste-à-temps (JIT) lors de l'exécution de l'application. Toutefois, il est également possible de compiler directement en code machine et d'ignorer le compilateur juste-à-temps en utilisant l'utilitaire Native Image Generator (Ngen.exe).
Cet utilitaire crée une image native à partir d'un assembly managé et l'installe dans le cache d'images natives sur l'ordinateur local. L'exécution de Ngen.exe sur un assembly permet de charger celui-ci plus rapidement, car ce programme restaure les structures de code et de données à partir du cache d'images natives au lieu de les générer de manière dynamique.

- ***Compilation à partir de la ligne de commande***

Pour compiler une application C# à partir de la ligne de commande, utilisez la commande ***csc***. Par exemple, pour compiler l'application « Hello, World » (Hello.cs) à partir de la ligne de commande, générer les informations de débogage et créer un fichier exécutable nommé Greet.exe, la commande est :

```
csc /debug+ /out:Greet.exe Hello.cs
```

Vérifiez que le fichier de sortie contenant le code compilé est spécifié avec un suffixe .exe. Si ce suffixe est omis, vous devez renommer le fichier avant de l'exécuter.

- ***Compilation à partir de Visual Studio***

Pour compiler une application C# à l'aide de Visual Studio, ouvrez le projet contenant l'application C#, puis cliquez sur ***Générer la solution*** dans le menu ***Générer***.

Par défaut, Visual Studio ouvre la configuration de débogage des projets. Cela signifie qu'une version déboguée de l'application est compilée.
Pour compiler une version Release sans informations de débogage, définissez la configuration de solution sur Release.

2.4.2. Exécution de l'application

Vous pouvez exécuter une application C# à partir de la ligne de commande ou à partir de Visual Studio.

- ***Exécution à partir de la ligne de commande***

Si la compilation de l'application réussit, un fichier exécutable (un fichier avec un suffixe .exe) est généré. Pour exécuter ce fichier depuis la ligne de commande, tapez le nom de l'application (avec ou sans le suffixe .exe).

- ***Exécution à partir de Visual Studio***

Pour exécuter l'application depuis Visual Studio, cliquez sur ***Exécuter sans débogage*** dans le menu **Déboguer**, ou appuyez sur CTRL+F5. Si l'application est une application console, une fenêtre de console s'affiche automatiquement, et l'application est exécutée. Une fois l'exécution terminée, le programme vous demande d'appuyer sur n'importe quelle touche pour continuer, et la fenêtre de console se ferme.

2.4.3. Débogage

- ***Exceptions et débogage JIT***

Si votre application renvoie une exception et que vous n'avez pas écrit de code pouvant gérer cette exception, le Common Language Runtime déclenche le débogage JIT (à ne pas confondre avec le compilateur JIT.)

En supposant que vous avez installé Visual Studio, une boîte de dialogue s'affiche pour vous permettre de déboguer l'application en utilisant le débogueur Visual Studio (environnement de développement Microsoft) ou le débogueur fourni avec le Kit de développement .NET Framework SDK.

Si vous avez accès à Visual Studio, il est recommandé de sélectionner le débogueur de Microsoft Development Environment.

- ***Définition de points d'arrêt et d'espions dans Visual Studio***

Vous pouvez utiliser le débogueur Visual Studio pour définir des points d'arrêt dans le code et examiner la valeur des variables.
Pour afficher un menu contenant de nombreuses options utiles, cliquez avec le bouton droit sur une ligne de code. Cliquez sur ***Insérer un point d'arrêt*** pour insérer un point d'arrêt sur cette ligne. Vous pouvez également insérer un point d'arrêt en cliquant dans la marge gauche. Cliquez une nouvelle fois pour supprimer ce point d'arrêt. Lorsque vous exécutez l'application en mode débogage, l'exécution s'arrête au niveau de cette ligne et vous pouvez examiner le contenu des variables.

La fenêtre Espion est utile pour contrôler la valeur des variables sélectionnées lors de l'exécution de l'application. Si vous saisissez le nom d'une variable dans la colonne ***Nom***, la valeur correspondante s'affiche dans la colonne ***Valeur***.
Lors de l'exécution de l'application, toutes les modifications de cette valeur s'affichent. Vous pouvez également modifier par écrasement la valeur d'une variable surveillée.

3. Utilisation des variables de type valeur

3.1. Système de types communs (CTS, Common System Type)

Chaque variable dispose d'un type de données qui détermine quelles valeurs qu'elle peut stocker. C# est un langage sécurisé au niveau des types : le compilateur C# s'assure que les types des valeurs stockées dans les variables sont toujours appropriés.
Le Common Language Runtime comprend un système de types communs qui définit un ensemble de types de données intégrés que vous pouvez utiliser pour définir vos variables.

3.1.1. Vue d'ensemble du système de types communs

Lorsque vous définissez une variable, vous devez lui attribuer un type de données approprié. Ce type de données détermine les valeurs autorisées pour cette variable, et ces valeurs autorisées déterminent les opérations possibles sur la variable.

Le système de types communs (CTS) fait partie intégrante du Common Language Runtime. Les compilateurs, les outils et le Runtime lui-même se partagent ce système. Il est le modèle définissant les règles suivies par le Runtime pour déclarer, utiliser et gérer les types. Il définit une structure qui permet l'intégration entre les langages, la sécurité des types et une exécution très performante du code.

3.1.2. Comparaison des types valeur et référence

- **Types valeur**

Les variables de type valeur contiennent directement leurs données. Chaque variable de type valeur dispose de son propre jeu de données ; les opérations sur une variable de ce type ne peuvent donc pas affecter une autre variable.

- **Types référence**

Les variables de type référence contiennent des références aux données qu'elles possèdent. Leurs données sont stockées dans un objet. Deux variables de type référence peuvent référencer le même objet. Les opérations sur une variable de ce type peuvent affecter l'objet référencé par une autre variable de type référence.

3.1.3. Comparaison des types valeur définis par l'utilisateur et des types valeur intégrés

Il existe différents types valeur : les types définis par l'utilisateur et les types intégrés. En C#, leur différence est minime, car les types définis par l'utilisateur peuvent être utilisés de la même façon que les types intégrés. La seule vraie différence entre les types de données intégrés et les types de données définis par l'utilisateur réside dans la possibilité d'utiliser des valeurs littérales pour les types intégrés.
Tous les types valeur contiennent des données, et celles-ci ne peuvent pas être ***null***.

3.1.4. Types simples

Les types valeur intégrés sont également appelés types de données de base, ou types simples. Des mots clés réservés permettent d'identifier les types simples. Ces mots clés réservés sont des alias des types struct prédéfinis.

Il est *impossible de distinguer* un type simple du type struct dont il est l'alias. Dans votre code, vous pouvez utiliser le mot clé réservé ou le type struct. Les exemples suivants illustrent les deux possibilités :

```
byte            // mot clé reservé

// ou

System.Byte    // type struct
```

```
int            // mot clé reservé

// ou

System.Int32   // type struct
```

Le tableau ci-dessus répertorie les mots clés réservés les plus courants et leur type ***struct*** équivalent.

Mots clés réservés	Alias du type struct
sbyte	System.SByte
byte	System.Byte
Short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.Int32
Long	System.Int64
ulong	System.UInt16
Char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

Tableau 4 - Mots clefs et leur équivalent en struct

3.2. Attribution de noms aux variables

Pour utiliser une variable, vous devez d'abord lui attribuer un nom approprié et significatif. Chaque variable possède un nom, également appelé *identificateur de variable*.

Respectez les conventions d'affectation de noms standard recommandées pour C#. Mémorisez les mots clés réservés en C# que vous ne pouvez pas utiliser comme noms de variables.

3.2.1. Règles et recommandations pour l'affectation de noms aux variables

Lorsque vous attribuez un nom à une variable, respectez les points suivants.

- ***Règles***

Les points suivants correspondent aux règles d'affectation de noms pour les variables C# :
- un nom de variable doit commencer par une lettre ou un trait de soulignement
- le premier caractère doit être suivi par des lettres, des chiffres ou un trait de soulignement
- les mots clés réservés sont proscrits
- l'utilisation d'un nom de variable non autorisé entraînera une erreur de compilation

- ***Recommandations***

Voici les recommandations à suivre pour l'affectation de noms de variables :
- éviter d'utiliser uniquement de lettres majuscules
- éviter les noms de variables commençant par un trait de soulignement
- éviter d'utiliser des abréviations

- utiliser la convention de noms de type casse Pascal

- ***Convention d'affectation de noms de type casse Pascal (PascalCasing)***

Pour utiliser la convention d'affectation de noms de type casse Pascal, mettez en majuscules le premier caractère de chaque mot. Utilisez cette convention pour les classes, les méthodes, les propriétés, les énumérations, les interfaces, les champs en lecture seule et les champs constants, les espaces de noms et les propriétés, comme illustré dans l'exemple suivant :

```
void InitializeData();
```

- ***Convention d'affectation de noms de type casse mixte (camelCasing)***

Pour utiliser la convention d'affectation de noms de type casse mixte, mettez en majuscules le premier caractère de chaque mot sauf pour le premier mot. Utilisez cette convention pour les variables qui définissent les champs et les paramètres, comme illustré dans l'exemple suivant :

```
int loopCountMax;
```

3.2.2. Mots clés en C#

Les mots clés sont réservés, ce qui signifie que vous ne pouvez pas les utiliser comme noms de variables en C#. L'utilisation d'un mot clé comme nom de variable entraînera une erreur de compilation.

Vous trouverez ci-dessous la liste des mots clés de C#. Rappelez-vous que vous ne pouvez pas les utiliser comme noms de variables.

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Tableau 5 - Mots clés de C#

3.3. Utilisation de types de données intégrés

Pour créer une variable, vous devez lui attribuer un nom, la déclarer et lui affecter une valeur, sauf si C# lui a déjà affecté une valeur automatiquement.

3.3.1. Déclaration de variables locales

Les variables qui sont déclarées dans les méthodes, les propriétés ou les indexeurs sont appelées variables locales. Généralement, vous déclarez une variable locale en spécifiant le type de données suivi du nom de la variable, comme illustré dans l'exemple suivant :

```
int itemCount;
```

Vous pouvez déclarer plusieurs variables dans une seule déclaration en utilisant un séparateur de type virgule, comme illustré dans l'exemple suivant :

```
int itemCount, employeeNumber;
```

En C#, vous ne pouvez pas utiliser de variables non initialisées. Le code suivant entraînera une erreur de compilation, car aucune valeur initiale n'a été attribuée à la variable ***loopCount*** :

```
int loopCount;
Console.WriteLine("{0}", loopCount);   // erreur
```

3.3.2. Attribution de valeurs aux variables

Les opérateurs d'assignation sont utilisés pour attribuer une nouvelle valeur à une variable. Pour attribuer une valeur à une variable déjà déclarée, utilisez l'opérateur d'assignation (=), comme illustré dans l'exemple suivant :

```
int employeeNumber;
employeeNumber = 23;
```

Vous pouvez aussi initialiser une variable lorsque vous la déclarez, comme illustré dans l'exemple suivant :

```
int employeeNumber = 23;
```

Vous pouvez utiliser l'opérateur d'assignation pour attribuer des valeurs aux variables de type caractère, comme illustré dans l'exemple suivant :

```
char middleInitial = 'J';
```

3.3.3. Assignation composé

- ***Ajouter une valeur à une variable est une opération très courante***

```
int itemCount;               // déclaration de la variable
itemCount = 2;               // affectation de la valeur 2
itemCount = itemCount + 40;  // incrémentation de 40
```

- ***La syntaxe abrégée est pratique***

Le code permettant d'incrémenter une variable fonctionne, mais il est quelque peu fastidieux. Vous devez écrire deux fois l'identificateur incrémenté. Pour les identificateurs simples, c'est rarement un problème, sauf si vous disposez de plusieurs identificateurs avec des noms très similaires. Toutefois, vous pouvez utiliser des expressions de complexité arbitraire pour désigner la valeur incrémentée, comme illustré dans l'exemple suivant :

```
items[(index + 1) % 32] = items[(index + 1) % 32] + 40;
```

Dans ces cas-là, si vous devez écrire la même expression deux fois, vous pouvez introduire facilement un bogue subtil. Heureusement, il existe une syntaxe abrégée qui permet d'éviter la duplication :

```
itemCount += 40;
items[(index + 1) % 32] += 40;
```

- ***Cette syntaxe abrégée fonctionne pour tous les opérateurs arithmétiques***

```
var += expression; // var = var + expression
var -= expression; // var = var - expression
var *= expression; // var = var * expression
var /= expression; // var = var / expression
var %= expression; // var = var % expression
```

3.3.4. Opérateurs courants

Les expressions sont créées à partir des *opérandes* et des *opérateurs*. Les opérateurs d'une expression indiquent les opérations à appliquer aux opérandes.

Certains des opérateurs les plus courants utilisés en C# sont décrits dans le tableau ci-dessous.

Type	Description	Exemple

Opérateurs d'assignation	Attribuent des valeurs aux variables en utilisant une simple assignation. Pour que l'assignation réussisse, la valeur située à droite de l'assignation doit être d'un type pouvant être converti implicitement dans le type de la variable située à gauche de l'assignation	`=`
		`*= /= %= += -=`
		`<<= >>=`
		`&= ^= |=`

Opérateurs logiques relationnels	Comparent deux valeurs.	`< > <= >= is`
Opérateurs logiques	Exécutent des opérations de bits sur les valeurs	
Opérateur conditionnel	Choisit entre deux expressions, selon une expression booléenne	`&& || ?:`
Opérateur d'incrément	Augmente la valeur de la variable d'une unité	`++`
Opérateur de décrément	Diminue la valeur de la variable d'une unité	`--`
Opérateurs arithmétiques	Exécutent des opérations arithmétiques standard	`+ - * / %`

Tableau 6 - Opérateurs les plus couramment utilisés en C#

3.4. Création de types de données définis par l'utilisateur

3.4.1. Types énumération (enum)

Les énumérateurs sont utiles lorsqu'une variable ne peut avoir qu'un ensemble spécifique de valeurs.

- ***Définition d'un type énumération***

Pour déclarer une énumération, utilisez le mot clé ***enum*** suivi du nom de la variable d'énumération et des valeurs initiales. Par exemple, l'énumération suivante définit trois constantes entières, appelées valeurs de l'énumérateur.

```
enum Color { Red, Green, Blue };
```

Par défaut, les valeurs de l'énumérateur commencent à 0. Dans l'exemple précédent, ***Red*** a la valeur 0, ***Green*** la valeur 1 et ***Blue***, la valeur 2.

- ***Utilisation d'un type énumération***

Vous pouvez déclarer une variable ***colorPalette*** de type ***Color*** en utilisant la syntaxe suivante :

```
Color colorPalette ;          // Déclare la variable
colorPalette = Color.Red ;    // Définit une valeur
 - ou -
colorPalette = (Color)0 ;     // Casting d'un type int en Color
```

- ***Affichage d'une valeur énumération***

Pour afficher une valeur énumération dans un format lisible, utilisez l'instruction suivante :

```
Console.WriteLine("{0}",colorPalette);
```

3.4.2. Types Structure (struct)

Vous pouvez utiliser des structures pour créer des objets qui se comportent comme des types valeur intégrés. Comme les structures sont stockées en ligne et ne sont pas allouées à un tas, la pression du garbage collection sur le système est moindre qu'avec les classes.

Dans le .NET Framework, tous les types de données simples tels que ***int***, ***float*** et ***double*** sont des structures intégrées.

- ***Définition d'un type structure***

Vous pouvez utiliser une structure pour regrouper plusieurs types arbitraires, comme illustré dans l'exemple suivant :

```
public struct Employee
{
      public string firstName;
      public int age;
}
```

Ce code définit un nouveau type appelé ***Employee*** qui se compose de deux éléments :
first name et ***age***.

- ***Utilisation d'un type structure***

Pour accéder aux éléments de la structure, vous devez utiliser la syntaxe suivante :

```
Employee companyEmployee;             // Déclare une variable
companyEmployee.firstName = "Jean";  // Définit sa valeur
companyEmployee.age = 23;
```

3.5. Conversion de types de données

3.5.1. Conversion implicite de types de données

La conversion d'un type de données ***int*** en un type de données ***long*** est implicite. Cette conversion réussit toujours et n'entraîne jamais de perte d'informations. L'exemple suivant illustre comment convertir la variable ***intValue*** d'un type ***int*** en un type ***long*** :

```
using System;

class Test
{
      static void Main( )
      {
            int  intValue  = 123;
            long longValue = intValue;
            Console.WriteLine("(long) {0} = {1}", intValue, longValue);
      }
}
```

3.5.2. Conversion explicite de types de données

La conversion explicite de données peut s'effectuer à l'aide d'une expression cast. L'exemple suivant illustre comment convertir la variable ***longValue*** d'un type de données ***long*** en un type de données ***int*** à l'aide d'une expression cast :

```
using System;

class Test
{
      static void Main( )
      {
            long longValue = Int64.MaxValue;
            int  intValue  = (int) longValue;
            Console.WriteLine("(int) {0} = {1}", longValue, intValue);
      }
}
```

Un dépassement s'étant produit dans cet exemple, le résultat est le suivant :

```
(int) 9223372036854775807 = -1
```

Pour éviter une telle situation, vous pouvez utiliser l'instruction ***checked*** pour lever une exception lorsqu'une conversion échoue, comme suit :

```
using System;

class Test
{
      static void Main( )
      {
          checked
          {
              long longValue = Int64.MaxValue;
              int intValue = (int) longValue;
              Console.WriteLine("(int) {0} = {1}", longValue, intValue);
          }
      }
}
```

4. Instructions et exceptions

4.1. Introduction aux instructions

Un programme se compose d'une suite d'instructions. Au moment de l'exécution, ces instructions sont exécutées les unes après les autres, dans l'ordre dans lequel elles figurent dans le programme, de gauche à droite et de haut en bas.

4.1.1. Blocs d'instructions

Lorsque vous développez des applications en C#, vous devez regrouper des instructions, tout comme vous le faites dans d'autres langages de programmation. Vous utilisez pour cela la même syntaxe qu'en C, C++ et Java, c'est-à-dire que vous placez les groupes d'instructions entre accolades : {et}.

- **Groupement d'instructions dans des blocs**

Un groupe d'instructions délimité par une accolade ouvrante et une accolade fermante constitue un bloc. Un bloc peut contenir une seule instruction ou un autre bloc imbriqué.

Chaque bloc définit une portée. Une variable déclarée dans un bloc est qualifiée de variable locale. La portée d'une variable locale s'étend de sa déclaration jusqu'à l'accolade fermante qui délimite la fin de son bloc. Le bon usage veut qu'une variable soit déclarée dans le bloc le plus profondément imbriqué, ainsi sa visibilité restreinte augmente la lisibilité du code.

- **Utilisation de variables dans des blocs d'instructions**

Dans C#, vous ne pouvez pas déclarer un nom de variable identique dans un bloc interne et externe. Par exemple, le code suivant n'est pas autorisé :

```
int i;
{
      int i; // Erreur : i déjà déclaré dans le bloc parent
      ...
}
```

Vous pouvez toutefois déclarer des variables portant le même nom dans des blocs *frères*. Les blocs frères sont délimités par le même bloc parent et sont imbriqués au même niveau.
En voici un exemple :

```
{
      int i;
      ...
}
...
{
      int i;
      ...
}
```

Les variables peuvent être déclarées n'importe où dans un bloc d'instructions, mais il est toutefois recommandé d'initialiser une variable au moment de sa déclaration.

4.1.2. Types d'instructions

La complexité de la logique d'un programme est proportionnelle à la complexité du problème qu'il doit résoudre. Pour résoudre des problèmes complexes, le contrôle du flux du programme doit être

structuré, ce qui peut être obtenu à l'aide de constructions ou d'instructions de haut niveau. Ces instructions peuvent être classées en trois catégories.

- ***Les instructions conditionnelles***

Les instructions ***if*** et ***switch*** sont des instructions conditionnelles (également appelées instructions de sélection). Ces instructions prennent des décisions en fonction de la valeur des expressions et exécutent des instructions de manière sélective en fonction de ces décisions.

- ***Les instructions d'itération***

Les instructions ***while***, ***do***, ***for*** et ***foreach*** s'exécutent tant qu'une condition particulière est vraie. Elles sont également qualifiées d'instructions de bouclage, du fait de leur caractère répétitif. Chacune de ces instructions correspond à un style particulier d'itération.

- ***Les instructions de saut***

Les instructions ***goto***, ***break*** et ***continue*** permettent de passer de façon inconditionnelle le contrôle à une autre instruction.

4.2.Utilisation des instructions conditionnelles

4.2.1. L'instruction if

La première instruction décisionnelle est l'instruction ***if***. Elle peut être associée à une clause ***else*** facultative, comme illustré ci-dessous :

```
if ( minute == 60 )
      minute = 0 ;
else
      minute++ ;
```

L'instruction ***if*** évalue une expression booléenne afin de déterminer la marche à suivre. Si l'expression booléenne a la valeur ***true***, la première instruction incorporée est exécutée. Si l'expression booléenne a la valeur ***false*** et qu'il y a une clause ***else***, le contrôle est passé à la seconde instruction incorporée.

L'instruction ***else if*** permet de traiter les instructions ***if*** en cascade.

```
if ( minute == 60 )
      minute = 0 ;
else if ( minute == -1 )
      minute-- ;
else
      minute++ ;
```

La construction ***else if*** permet un nombre indéfini de branchements. Cependant, les instructions contrôlées par une instruction ***if*** en cascade étant mutuellement exclusives, une seule instruction de l'ensemble de constructions ***else if*** sera exécutée.

4.2.2. L'instruction switch

L'instruction ***switch*** offre une méthode de traitement des conditions complexes plus élégante que les instructions ***if*** imbriquées. Elle se compose de plusieurs blocs de case, chaque bloc spécifiant une constante unique et une étiquette ***case*** associée.

Un bloc ***switch*** peut contenir des déclarations. La portée d'une variable locale ou d'une constante déclarée dans un bloc ***switch*** s'étend depuis sa déclaration jusqu'à la fin du bloc ***switch***, comme l'illustre l'exemple reproduit sur la diapositive.

- **Exécution des instructions switch**

Une instruction ***switch*** s'exécute comme suit :

1 - Si l'une des constantes spécifiées dans une étiquette ***case*** est égale à la valeur de l'expression ***switch***, la liste des instructions qui suit l'étiquette ***case*** correspondante est exécutée.

2 - Si aucune constante ***case*** n'est égale à la valeur de l'expression ***switch*** et si l'instruction ***switch*** contient une étiquette ***default***, la liste des instructions qui suit l'étiquette ***default*** est exécutée.

3 - Si aucune constante ***case*** n'est égale à la valeur de l'expression ***switch*** et si l'instruction ***switch*** ne contient pas d'étiquette ***default***, le contrôle est transféré à la fin de l'instruction ***switch***.

```
switch (atout)
{
   case Suite.Pique :
      couleur = "Noir";
      break;
   case Suite.Carreau :
      couleur = "Rouge";
      break;
   default :
      couleur = "ERREUR";
      break;
}
```

L'instruction ***switch*** évalue uniquement les types d'expressions suivants : tous les entiers, ***char***, ***enum*** et ***string***. Vous pouvez toutefois évaluer d'autres types d'expressions à l'aide de l'instruction ***switch*** du moment qu'il existe précisément une conversion implicite définie par l'utilisateur du type non autorisé vers les types autorisés.

- **Utilisation de l'instruction break dans les instructions switch**

Contrairement à ce qui se passe dans les langages Java, C ou C++, les instructions C# associées à une ou plusieurs étiquettes ***case*** ne doivent pas passer silencieusement ou se brancher sur l'étiquette ***case*** suivante. Un *passage silencieux* se produit lorsque l'exécution se poursuit sans générer d'erreur. Autrement dit, vous devez vous assurer que la dernière instruction associée à un groupe d'étiquettes ***case*** ne permet pas au flux de contrôle d'atteindre le groupe suivant d'étiquettes ***case***.

Pour respecter cette règle, appelée *règle de non-passage*, vous pouvez recourir à l'une des instructions suivantes : ***break*** (probablement la plus courante), ***goto*** (très rare), ***return***, ***throw*** ou une boucle infinie.

4.3. Utilisation des instructions d'itération

4.3.1. L'instruction while

Parmi les instructions d'itération, ***while*** est la plus simple à utiliser. Elle exécute une instruction incorporée *tant que* l'expression booléenne est vraie.

Une instruction ***while*** s'exécute comme suit :

1 - L'expression booléenne qui contrôle l'instruction ***while*** est évaluée.
2 - Si l'expression booléenne a la valeur ***true***, l'instruction incorporée est exécutée. À la fin de cette exécution, le contrôle repasse implicitement au début de l'instruction ***while*** et l'expression booléenne est de nouveau analysée.

3 - Si l'expression booléenne a la valeur ***false***, le contrôle est transféré à la fin de l'instruction ***while***. Par conséquent, le programme réexécute l'instruction incorporée tant que l'expression booléenne a la valeur ***true***.

L'expression booléenne étant testée au début de la boucle ***while***, il est possible que l'instruction incorporée ne soit jamais exécutée.

```
int i = 0;
while (i < 10)
{
      Console.WriteLine(i);
      i++;
}

// affiche : 0 1 2 3 4 5 6 7 8 9
```

4.3.2. L'instruction do

L'instruction do est toujours accompagnée d'une instruction ***while***. Elle est similaire à l'instruction ***while***, à cette différence près que l'analyse de l'expression booléenne qui détermine la poursuite ou la sortie de la boucle s'effectue en fin de boucle, et non pas au début. Cela signifie que, contrairement à une instruction ***while*** qui est exécutée zéro ou plusieurs fois, une instruction do est exécutée une au moins fois.

Une instruction ***do*** s'exécute comme suit :

1 - L'instruction incorporée est exécutée.
2 - Cela fait, l'expression booléenne est évaluée.
3 - Si l'expression booléenne a la valeur ***true***, le contrôle est transféré au début de l'instruction ***do***.
4 - Si l'expression booléenne a la valeur ***false***, le contrôle est transféré à la fin de l'instruction ***do***.

```
int i = 0;
do
{
      Console.WriteLine(i);
      i++;
}
while (i < 10)

// affiche : 0 1 2 3 4 5 6 7 8 9
```

4.3.3. L'instruction for

Dans L'instruction ***for***, le code d'actualisation se trouve au début de la boucle où il est plus difficilement oublié. La syntaxe de l'instruction ***for*** est la suivante :

```
for ( initialiseur ; condition ; actualisation )
      instruction-incorporée
```

La syntaxe de l'instruction ***for*** est pratiquement identique à celle de l'instruction ***while***, comme l'illustre l'exemple suivant :

```
initialiseur
while ( condition )
{
      instruction-incorporée
      actualisation
}
```

Comme dans toutes les instructions de répétition, la condition d'un bloc ***for*** doit être une expression booléenne qui tient lieu de condition de poursuite de la boucle et non pas de condition de fin.

- ***Exemple***

Comme pour les instructions ***while*** et ***do***, vous pouvez utiliser une seule instruction incorporée ou un bloc d'instructions, comme dans les exemples suivants :

```
for ( int i = 0 ; i < 10 ; i++ )
      Console.WriteLine(i);

// affiche : 0 1 2 3 4 5 6 7 8 9

for ( int i = 0 ; i < 10 ; i++ ) {
      Console.WriteLine(i);
      Console.WriteLine(i-9);
}

// affiche : 0 -9 1 -8 2 -7 3 -6 4 -5 5 -4 6 -3 7 -2 8 -1 9 0
```

- ***Déclaration de variables***

Il existe une petite différence entre les instructions ***while*** et ***for*** : la portée d'une variable déclarée dans le code d'initialisation d'une instruction ***for*** est limitée au bloc ***for***. Ainsi, l'exemple suivant génère une erreur de compilation :

```
for ( int i = 0; i < 10; i++ )
      Console.WriteLine(i);

Console.WriteLine(i); // Erreur : i n'est plus dans la portée
```

Notez, par ailleurs, que vous ne pouvez pas déclarer dans un bloc ***for*** une variable portant le même nom qu'une autre variable placée dans un bloc externe. Ainsi, l'exemple suivant génère une erreur de compilation :

```
int i;
for (int i = 0; i < 10; i++) // Erreur : i est déjà dans la portée
```

En outre, vous pouvez initialiser deux variables ou plus dans le code d'initialisation d'une instruction ***for*** de la manière suivante :

```
for (int i = 0, j = 0; i < 10; i++)
```

Les variables doivent toutefois être du même type. Par conséquent, l'exemple suivant n'est pas autorisé :

```
for (int i = 0, long j = 0; i < 10; i++)
```

Pour mettre plusieurs instructions dans le code d'actualisation d'une instruction ***for***, il faut les séparer par une virgule ou plusieurs virgules, comme suit :

```
for (int i = 0, j = 0; i < 10 ; i++, j++)
```

L'instruction ***for*** est adaptée aux situations dans lesquelles le nombre d'itérations est connu. Elle est également particulièrement appropriée pour modifier chaque élément d'un tableau.

4.3.4. L'instruction foreach

Les collections sont des entités logicielles qui servent à collecter d'autres entités logicielles. Nous pouvons faire une analogie avec un grand livre qui forme une collection de comptes, ou avec une maison qui est une collection de pièces.

Microsoft .NET Framework offre une classe de collection simple nommée ArrayList. Elle permet de créer une variable de collection et d'ajouter des éléments à la collection. Considérons une collection comme une liste.

C# fournit l'instruction ***foreach***, qui permet de parcourir chaque élément d'une collection sans recourir à une longue liste d'instructions. Plutôt que d'extraire explicitement chaque élément d'une collection avec une syntaxe spécifique à la collection, l'instruction ***foreach*** permet d'aborder le problème différemment. Vous demandez en fait à la collection de présenter ses éléments, les uns à la suite des autres. Ce n'est plus l'instruction incorporée qui est apportée à la collection, c'est la collection qui est apportée à l'instruction incorporée.

Examinons par exemple le code suivant :

```
using System.Collections;

ArrayList numbers = new ArrayList( );    // numbers est une collection

for (int i = 0; i < numbers.Count; i++) {
      int number = (int)numbers[i];
      Console.WriteLine(number);
}

// L'instruction for précédente peut être réécrite
// comme suit avec une instruction foreach

foreach (int number in numbers)
      Console.WriteLine(number);
```

4.4. Utilisation des instructions de saut

4.4.1. L'instruction goto

L'instruction ***goto*** est l'instruction de saut la plus primitive du langage C#. Elle permet de passer le contrôle d'exécution à une instruction marquée par une étiquette. L'étiquette doit exister et sa portée doit s'étendre à l'instruction ***goto***. Plusieurs instructions ***goto*** peuvent transférer le contrôle d'exécution vers la même étiquette.

L'instruction ***goto*** peut transférer le contrôle à l'extérieur d'un bloc, mais ne peut jamais le transférer dans un bloc. Le but de cette restriction est d'éviter de sauter une initialisation. Cette même règle est appliquée en C++ et dans d'autres langages
.
L'instruction ***goto*** et l'étiquette cible peuvent être très éloignées l'une de l'autre dans le code. Cette distance pouvant obscurcir la logique du flux de contrôle, la plupart des guides de programmation recommandent de ne pas utiliser les instructions ***goto***.

```
if (number % 2 == 0) goto Even;
Console.WriteLine("odd");
goto End;

Even:
Console.WriteLine("even");
End:;

// affiche : even
```

Les instructions ***goto*** sont recommandées dans les deux cas suivants : dans les instructions ***switch*** et pour sortir d'une boucle imbriquée.

4.4.2. Les instructions break et continue

Une instruction ***break*** permet de sortir de la plus proche instruction ***switch***, ***while***, ***do***, ***for*** ou ***foreach*** qui l'entoure. Une instruction ***continue*** provoque une nouvelle itération de la plus proche instruction ***switch***, ***while***, ***do***, ***for*** ou ***foreach*** qui l'entoure.

Les instructions ***break*** et ***continue*** ne sont pas très différentes d'une instruction ***goto***, qui elle peut aisément obscurcir la logique du flux de contrôle.

```
int i = 0;
while (true)
{
   Console.WriteLine(i);
   i++;
   if (i < 10)
      continue;
   else
      break;
}

// affiche : 0 1 2 3 4 5 6 7 8 9
```

4.5. Gestion des exceptions fondamentales

4.5.1. Objets exception

Voici un code d'erreur de programmation utilisé dans du code de gestion des erreurs de type procédure

```
enum ErrorCode { SecurityError=-1, IOError=-2, OutOfMemoryError=-3, ... }
```

Ce type de code d'erreur ne permet pas de fournir aisément les informations qui aideront à la réparer. Ainsi, la génération de ***IOError*** ne vous renseigne pas sur le type précis de l'erreur.
S'agit-il d'une tentative d'écriture sur un fichier en lecture seule ou sur un fichier inexistant, ou d'un disque corrompu ? Sur quel fichier a lieu la tentative de lecture ou d'écriture ?

Pour remédier à ce manque d'information sur l'erreur générée, .NET Framework a défini un ensemble de classes d'exceptions.

Toutes les exceptions sont héritées de la classe nommée ***Exception***, qui fait partie du Common Language Runtime. L'image ci-dessus présente la hiérarchie de ces exceptions.

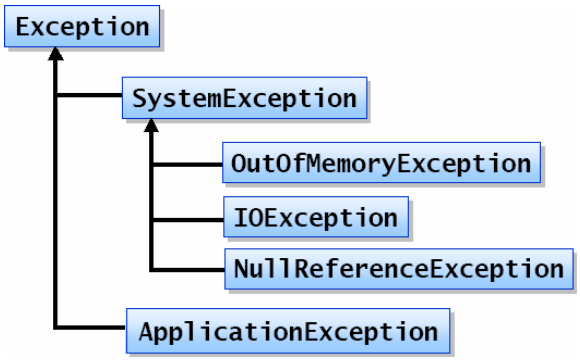

Figure 4 - Hiérarchie des exceptions

Les classes d'exceptions offrent les avantages suivants :

- Les messages d'erreur ne sont plus représentés par des valeurs d'entier ou des énumérations

Les valeurs d'entier de programmation, telles que ***-3*** ont été supprimées. Elles ont été remplacées par les classes d'exceptions, telle que ***OutOfMemoryException***. Chaque classe d'exceptions peut résider dans son propre fichier source, et être dissociée de toutes les autres classes d'exceptions.

- Des messages d'erreur significatifs sont générés.

Chaque classe d'exceptions décrit clairement l'erreur qu'elle représente. Une classe nommée ***OutOfMemoryException*** remplace efficacement un –3. Chaque classe d'exceptions peut également contenir des informations qui lui sont propres. Ainsi, une classe ***FileNotFoundException*** peut contenir le nom du fichier introuvable.

4.5.2. Utilisation des blocs try et catch

La programmation orientée objet offre une solution structurée de gestion des erreurs, sous la forme de blocs ***try*** et ***catch***. L'idée est de séparer physiquement les instructions essentielles du programme qui gèrent son déroulement normal, des instructions de gestion des erreurs. Par conséquent, les parties de code susceptibles de lever des exceptions sont placées dans un bloc ***try***, et le code de traitement de ces exceptions est placé dans un bloc ***catch*** distinct.

La syntaxe d'un bloc ***catch*** est la suivante :

```
catch ( identificateur-de-type-de-classe ) { ... }
```

Le type de classe doit être soit ***System.Exception***, soit un type dérivé de ***System.Exception***.

L'identificateur, qui est facultatif, est une variable locale en lecture seule dans la portée du bloc ***catch***.

```
catch (Exception caught) {
...
}
Console.WriteLine(caught);    // Erreur de compilation
                              // caught n'est plus dans la portée
```

```
try {
      Console.WriteLine("Entrez un nombre");
      int i = int.Parse(Console.ReadLine());
}
catch (OverflowException caught)
{
      Console.WriteLine(caught);
}
```

L'exemple de code ci-dessus illustre l'utilisation des instructions ***try*** et ***catch***. Le bloc ***try*** entoure une expression susceptible de générer une exception. Si l'exception est levée, le Runtime interrompt l'exécution et cherche un bloc ***catch*** pouvant intercepter cette exception (en fonction de son type). Si le Runtime ne trouve pas de bloc ***catch*** approprié dans la fonction la plus proche, il déroule la pile des appels pour trouver l'appel de fonction. Si le bloc ***catch*** approprié ne s'y trouve pas, il recherche la fonction qui a émis l'appel, et ainsi de suite jusqu'à ce qu'il trouve un bloc ***catch*** approprié (Ou jusqu'à ce qu'il atteigne la fin de ***Main***, auquel cas le programme s'arrête). Si le Runtime trouve un bloc ***catch***, il considère que l'exception est interceptée, et il reprend l'exécution du code à partir du corps du bloc ***catch***.

4.5.3. Blocs catch multiples

Un bloc de code à l'intérieur d'une construction ***try*** peut comporter plusieurs instructions. Chaque instruction peut lever une ou plusieurs classes d'exceptions. Comme il existe un grand nombre de classes d'exceptions, vous pouvez utiliser plusieurs blocs ***catch***, qui interceptent chacun un type d'exception particulier.

Une exception est interceptée uniquement sur la base de son type. Le runtime intercepte automatiquement les objets exception d'un type particulier dans un bloc ***catch*** du type correspondant.

Considérons le code suivant pour mieux appréhender le fonctionnement d'un bloc ***try-catch*** multiple :

```
1.  try {
2.     Console.WriteLine("Entrez le premier nombre");
3.     int i = int.Parse(Console.ReadLine());
4.     Console.WriteLine("Entrez le second nombre");
5.     int j = int.Parse(Console.ReadLine());
6.     int k = i / j;
7.  }
8.  catch (OverflowException caught)
9.  {
10.    Console.WriteLine(caught);
11. }
12. catch(DivideByZeroException caught)
13. {
14.    Console.WriteLine(caught);
15. }
```

La ligne 3 initialise ***int i*** avec une valeur lue à partir de la console. Cette opération est susceptible de lever un objet exception de la classe ***OverflowException***. Si c'est le cas, les lignes 4, 5 et 6 ne sont pas exécutées. L'exécution séquentielle normale est interrompue et le contrôle d'exécution est transféré sur le premier bloc ***catch*** capable d'intercepter cette exception. Dans cet exemple, il s'agit du bloc ***catch*** de la ligne 8.

En revanche, si les lignes 3 à 5 ne lèvent aucune exception, l'exécution séquentielle se poursuit normalement jusqu'à la ligne 6. Cette ligne est susceptible de lever un objet exception de la classe ***DivideByZeroException***. Si c'est le cas, le flux de contrôle passe au bloc ***catch*** à la ligne 12, exécute ce bloc.

Si aucune des instructions du bloc ***try*** ne lève d'exception, le flux de contrôle atteint la fin de ce bloc. Remarquez que le flux de contrôle n'entre dans un bloc ***catch*** que si une exception a été levée.

Vous pouvez écrire les instructions dans un bloc ***try*** sans vous inquiéter de savoir si une instruction antérieure dans ce bloc risque d'échouer. Si une instruction antérieure lève une exception, le flux de contrôle n'atteindra pas physiquement les instructions qui la suivent dans le bloc ***try***.

4.6. Levée d'exceptions

4.6.1. L'instruction throw

- **Exceptions définies par le système**

Pour lever une exception, le Runtime exécute une instruction ***throw*** et lève une exception définie par le système. L'exécution séquentielle normale du programme est interrompue immédiatement et le contrôle d'exécution est transféré sur le premier bloc ***catch*** capable de gérer l'exception, en fonction de sa classe.

- **Comment lever vos propres exceptions**

Vous pouvez recourir à l'instruction ***throw*** pour lever vos propres exceptions :

```
if (minute < 1 || minute >= 60) {
      string fault = minute + " n'est pas une minute valide";
      throw new InvalidTimeException(fault);
      // !!Pas atteint!!
}
```

Dans cet exemple, l'instruction ***throw*** lève une exception définie par l'utilisateur, ***InvalidTimeException***, si le temps analysé ne correspond pas à une valeur de temps correcte.
En règle générale, les exceptions sont créées avec une chaîne de message significatif en paramètre. Ce message est affiché ou enregistré au moment où l'exception est détectée. Il est également recommandé de lever une classe d'exceptions appropriée.

4.6.2. La clause finally

C# fournit la clause ***finally*** pour entourer les instructions qui doivent être exécutées, quoi qu'il arrive. Autrement dit, si le contrôle sort normalement d'un bloc ***try*** parce que le flux de contrôle a atteint la fin de ce bloc, les instructions du bloc ***finally*** sont exécutées. De même, si le contrôle d'exécution sort d'un bloc ***try*** en raison d'une instruction ***throw*** ou d'une instruction de saut, telle que ***break***, ***continue*** ou ***goto***, les instructions de la clause ***finally*** sont exécutées.

Le bloc ***finally*** est utile dans deux circonstances : pour éviter la duplication d'instructions et pour libérer des ressources après la levée d'une exception.

```
try {
      ...
}
catch {
      ...
}
finally {
      instructions
}
```

5. Méthodes et paramètres

5.1. Utilisation des méthodes

5.1.1. Définition des méthodes

Une méthode est un groupe d'instructions C# qui ont été rassemblées et auxquelles un nom a été attribué. La plupart des langages de programmation modernes partagent un concept similaire ; vous pouvez vous représenter une méthode comme une fonction, une sous-routine, une procédure ou un sous-programme.

- **Syntaxe pour la définition des méthodes**

Pour créer une méthode, utilisez la syntaxe suivante :

```
static void NomMéthode( )
{
    // corps de la méthode
}
```

- **Exemples de méthodes**

L'exemple suivant montre comment créer une méthode appelée ***ExampleMethod*** dans la classe ***ExampleClass*** :

```
using System;
class ExampleClass
{
    static void ExampleMethod( )
    {
        Console.WriteLine("Méthode ExampleMethod");
    }

    static void Main( )
    {
        Console.WriteLine("Méthode Main");
    }
}
```

Le code contient trois méthodes :

- ***Main***
- ***WriteLine***
- ***ExampleMethod***

La méthode ***Main*** est le point d'entrée de l'application. La méthode ***WriteLine*** fait partie intégrante de Microsoft® .NET Framework. C'est une méthode statique de la classe ***System.Console***. La méthode ***ExampleMethod*** appartient à la classe ***ExampleClass***.

Dans le langage C#, toutes les méthodes appartiennent à une classe. Ce n'est pas le cas des langages de programmation tels que C, C++ et Microsoft Visual Basic®, qui autorisent l'utilisation de fonctions et de sous-routines globales.

- **Création de méthodes**

Lorsque vous créez une méthode, vous devez spécifier les éléments suivants :

- Nom :

Vous ne pouvez pas attribuer à une méthode un nom identique à celui d'une variable, d'une constante ou de tout élément qui n'est pas une méthode déclarée dans la classe. Il est possible d'utiliser comme nom de méthode tout identificateur C# autorisé. Ce nom respecte la casse.

- Liste des paramètres :

Le nom de la méthode est suivi par la liste des paramètres de la méthode.
Cette liste figure entre parenthèses. Ces parenthèses doivent être utilisées même si la liste ne contient aucun paramètre, comme le montrent les exemples du dessus.

- Corps de la méthode :

À la suite des parenthèses, se trouve le corps de la méthode. Vous devez placer le corps de la méthode entre accolades (***{*** et ***}***), même si une seule instruction est utilisée.

5.1.2. Appel de méthodes

Lorsqu'une méthode est définie, vous pouvez l'appeler à partir de la classe dans laquelle elle a été définie et à partir d'autres classes.

Pour appeler une méthode, utilisez le nom de la méthode suivi de la liste des paramètres entre parenthèses. Les parenthèses sont requises même si la méthode que vous appelez ne contient aucun paramètre, comme le montre l'exemple suivant.

```
MethodName( );
```

Dans l'exemple suivant, le programme commence au début de la méthode ***Main*** de la classe ***ExampleClass***. La première instruction affiche « Le programme démarre ». La deuxième instruction de la méthode ***Main*** est l'appel de la méthode ***ExampleMethod***. Le flux de contrôle passe à la première instruction de la méthode ***ExampleMethod*** et le message « Hello, world » s'affiche. À la fin de la méthode, le contrôle passe à l'instruction qui suit immédiatement l'appel de la méthode, c'est-à-dire à l'instruction qui affiche le message « Le programme prend fin ».

```
using System;
class ExampleClass
{
   static void ExampleMethod( )
   {
      Console.WriteLine("Hello, world");
   }

   static void Main( )
   {
      Console.WriteLine("Le programme démarre");
      ExampleMethod( );
      Console.WriteLine("Le programme prend fin");
   }
}
```

5.1.3. Utilisation de l'instruction return

Vous pouvez utiliser l'instruction ***return*** pour qu'une méthode retourne immédiatement à l'appelant. Sans instruction ***return***, l'exécution retourne à l'appelant lorsque la dernière instruction de la méthode est atteinte.

Par défaut, une méthode retourne à son appelant lorsque la fin de la dernière instruction du bloc de code est atteinte. Si vous souhaitez qu'elle retourne immédiatement à l'appelant, utilisez l'instruction ***return***.

Dans l'exemple suivant, la méthode affiche « Hello » et retourne immédiatement à son appelant :

```
static void ExampleMethod( )
{
   Console.WriteLine("Hello");
   return;
   Console.WriteLine("World");
}
```

Il n'est pas très utile d'utiliser l'instruction ***return*** de cette façon, car l'appel final de la méthode ***Console.WriteLine*** n'est jamais exécuté.

5.1.4. Retour de valeurs

- ***Déclaration de méthodes avec un type de retour non void***

Pour déclarer une méthode de façon à ce qu'elle retourne une valeur à l'appelant, remplacez le mot clé ***void*** par le type de la valeur à retourner.

- ***Ajout d'instructions return***

Le mot clé ***return*** suivi d'une expression termine la méthode immédiatement et retourne l'expression comme valeur de retour de la méthode.

L'exemple suivant montre comment déclarer une méthode nommée ***TwoPlusTwo*** qui retourne la valeur 4 à la méthode ***Main*** lorsque la méthode ***TwoPlusTwo*** est appelée :

```
class ExampleReturningValue
{
   static int TwoPlusTwo( )
   {
      int a,b;
      a = 2;
      b = 2;
      return a + b;
   }
}
```

Notez que la valeur retournée est un entier (***int***). ***int*** est en effet le type de retour de la méthode. Lorsque la méthode est appelée, la valeur 4 est retournée.

Si vous déclarez une méthode avec un type non ***void***, vous devez ajouter au moins une instruction ***return***. Le compilateur vérifie que chaque méthode non ***void*** retourne dans tous les cas une valeur à la méthode appelante.

5.2.Utilisation des paramètres

5.2.1. Déclaration et appel de paramètres

Les paramètres permettent aux informations d'être passées à l'intérieur et à l'extérieur d'une méthode. Lorsque vous définissez une méthode, vous pouvez inclure une liste de paramètres entre parenthèses, précédée du nom de la méthode.

- **Déclaration des paramètres**

Chaque paramètre se caractérise par un type et un nom. Pour déclarer des paramètres, vous devez placer les déclarations de paramètres à l'intérieur des parenthèses qui suivent le nom de la méthode. La syntaxe utilisée pour déclarer les paramètres est similaire à celle utilisée pour déclarer des variables locales, sauf que vous séparez chaque déclaration de paramètre par une virgule au lieu d'un point-virgule.

L'exemple suivant montre comment déclarer une méthode avec des paramètres.

```
static void MethodWithParameters(int n, string y)
{
   // ...
}
```

Cet exemple montre comment déclarer la méthode ***MethodWithParameters*** avec deux paramètres : *n* et *y*. Le premier paramètre est du type ***int*** (entier) et le second du type ***string*** (chaîne). Notez que des virgules séparent chaque paramètre dans la liste des paramètres.

- **Appel de méthodes avec des paramètres**

Le code appelant doit fournir les valeurs des paramètres lorsque la méthode est appelée.

Le code suivant illustre une façon d'appeler une méthode avec des paramètres. Les valeurs des paramètres sont trouvées et placées dans les paramètres *n* et *y* au début de l'exécution de la méthode ***MethodWithParameters***.

```
MethodWithParameters(2, "Hello, World");
```

5.2.2. Mécanismes de passage de paramètres

Il existe trois façons de passer des paramètres :

- ***Par valeur (in) :***

Les paramètres par valeur sont parfois appelés *paramètres d'entrée* parce que les données peuvent être transférées à l'intérieur de la méthode mais non à l'extérieur.

- ***Par référence (in out) :***

Les paramètres par référence sont parfois appelés *paramètres d'entrée/sortie* parce que les données peuvent être transférées à l'intérieur de la méthode puis à nouveau à l'extérieur.

- ***Par sortie (out) :***

Les paramètres de sortie sont appelés ainsi parce que les données peuvent être transférées à l'extérieur de la méthode, mais non à l'intérieur.

5.2.3. Passage par valeur

- **Définition des paramètres par valeur**

La définition la plus simple d'un paramètre est un nom de type suivi d'un nom de variable. On parle de *paramètre par valeur*. Lorsque la méthode est appelée, un nouvel emplacement de stockage est créé pour chaque paramètre par valeur, et les valeurs des expressions correspondantes y sont copiées.

À l'intérieur de la méthode, vous pouvez écrire du code qui modifie la valeur du paramètre. Ceci n'aura aucun effet sur les variables situées en dehors de l'appel de la méthode.

Dans l'exemple suivant, la variable ***x*** à l'intérieur de la méthode ***AddOne et*** peut être modifiée dans la méthode ***AddOne***, mais cette modification n'a pas d'effet sur la variable ***k***.

```
static void AddOne(int x)
{
   x++;
}

static void Main( )
{
   int k = 6;
   AddOne(k);
   Console.WriteLine(k); // Affiche la valeur 6, mais pas 7
}
```

5.2.4. Passage par référence

- **Définition des paramètres par référence**

Un paramètre par référence fait référence à un emplacement dans la mémoire. Contrairement à un paramètre par valeur, un paramètre par référence ne crée pas de nouveaux emplacements de stockage. Au lieu de cela, il représente le même emplacement dans la mémoire que la variable qui est fournie dans l'appel de la méthode.

- **Modification des valeurs des paramètres par référence**

Si vous modifiez la valeur d'un paramètre par référence, la variable fournie par l'appelant est également modifiée, car elles font toutes deux référence au même emplacement dans la mémoire. L'exemple suivant montre comment la modification du paramètre par référence entraîne également celle de la variable :

```
static void AddOne(ref int x)
{
   x++;
}

static void Main( )
{
   int k = 6;
   AddOne(ref k);
   Console.WriteLine(k); // Affiche la valeur 7
}
```

- ***Affectation des paramètres avant l'appel de la méthode***

Un paramètre ***ref*** doit être définitivement assigné au point de l'appel ; en d'autres termes, le compilateur doit pouvoir s'assurer qu'une valeur a bien été assignée avant que l'appel ne soit effectué. L'exemple précédent montre comment vous pouvez initialiser des paramètres par référence avant d'appeler une méthode.

L'exemple suivant montre ce qui se produit si un paramètre par référence ***k*** n'est pas initialisé avant que sa méthode ***AddOne*** ne soit appelée :

```
int k;
AddOne(ref k);
Console.WriteLine(x);
```

Le compilateur C# rejette ce code et affiche le message d'erreur suivant :
« Utilisation d'une variable locale non assignée '*k*' ».

5.2.5. Paramètres de sortie

- ***Définition des paramètres de sortie***

Les paramètres de sortie sont identiques aux paramètres par référence, sauf qu'ils transfèrent des données à l'extérieur de la méthode et non pas à l'intérieur. Tout comme un paramètre par référence, un paramètre de sortie est une référence à un emplacement de stockage fourni par l'appelant. Toutefois, il n'est pas nécessaire d'assigner une valeur à la variable fournie pour le paramètre ***out*** avant que l'appel ne soit effectué ; la méthode suppose que le paramètre n'a pas été initialisé à l'entrée.

Les paramètres de sortie sont utiles lorsque vous voulez retourner des valeurs à partir d'une méthode au moyen d'un paramètre sans assigner de valeur initiale à ce dernier.

- ***Utilisation des paramètres de sortie***

Pour déclarer un paramètre de sortie, utilisez le mot clé ***out*** avant le type et le nom, comme le montre l'exemple suivant :

```
static void OutDemo(out int p)
{
   // ...
}
```

Tout comme c'est le cas avec le mot clé ***ref***, le mot clé ***out*** assigne un seul paramètre, et chaque paramètre ***out*** doit être marqué séparément.
Lorsque vous appelez une méthode avec un paramètre ***out***, placez le mot clé ***out*** avant la variable à passer, comme dans l'exemple suivant :

```
int n;
OutDemo(out n);
```

Dans le corps de la méthode appelée, aucune supposition de départ n'est faite sur le contenu du paramètre de sortie. Il est traité comme une variable locale non assignée. Une valeur doit être assignée au paramètre ***out*** à l'intérieur de la méthode.

6. Tableaux

6.1. Vue d'ensemble des tableaux

6.1.1. Qu'est ce qu'un tableau ?

Les tableaux sont des séquences de données de même type. Vous pouvez accéder à un élément individuel d'un tableau en utilisant sa position d'entier, qui s'appelle un index.
Les tableaux autorisent l'accès aléatoire. Les éléments d'un tableau sont situés dans la mémoire contiguë. Cela signifie qu'un programme peut accéder aussi rapidement à tous les éléments d'un tableau.

6.1.2. Notation de tableau en C#

Vous utilisez la même notation pour déclarer un tableau ou une simple variable. Spécifiez d'abord le type, puis le nom de la variable suivi d'un point-virgule. Vous déclarez le type de la variable comme un tableau à l'aide de crochets. De nombreux autres langages de programmation, comme C et C++, utilisent également les crochets pour déclarer un tableau.

En C#, la notation de tableau est identique à celle employée par C et C++, à ces différences près :

- Vous ne pouvez pas écrire de crochets à droite du nom de la variable.
- Vous ne spécifiez pas la taille du tableau lorsque vous déclarez une variable tableau.

Les exemples suivants illustrent des notations autorisées et interdites en C# :

```
type[ ]name;    // Autorisé
type name[ ];   // Interdit en C#
type[4] name;   // Également interdit en C#
```

6.1.3. Rang de tableau

Pour déclarer une variable tableau à une dimension, vous utilisez des crochets simples comme illustré dans l'exemple précédent. Un tableau de ce type s'appelle également tableau de rang 1, car un index entier est associé à chaque élément du tableau.

Pour déclarer un tableau à deux dimensions, vous utilisez une virgule à l'intérieur des crochets, comme illustré sur la diapositive. Un tableau de ce type s'appelle également tableau de rang 2, car deux index entiers sont associés à chaque élément du tableau. Cette notation s'étend simplement comme suit : chaque virgule supplémentaire spécifiée entre les crochets incrémente d'une unité le rang du tableau.

Vous n'incluez pas la longueur des dimensions dans la déclaration d'une variable tableau.

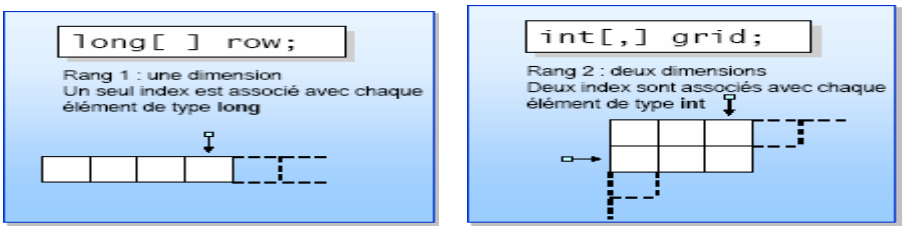

Figure 5 - Rangs de tableaux

6.1.4. Accès aux éléments d'un tableau

Pour accéder à un élément à l'intérieur d'un tableau de rang 1, utilisez un index entier. Pour accéder à un élément à l'intérieur d'un tableau de rang 2, utilisez deux index entiers séparés par une virgule. Pour

accéder à un élément à l'intérieur d'un tableau de rang *n*, utilisez *n* index entiers séparés par des virgules.

Les index de tableau (de tout rang) démarrent à zéro. Pour accéder au premier élément à l'intérieur d'une ligne, utilisez l'expression :

```
row[0]
```

Plutôt que l'expression :

```
row[1]
```

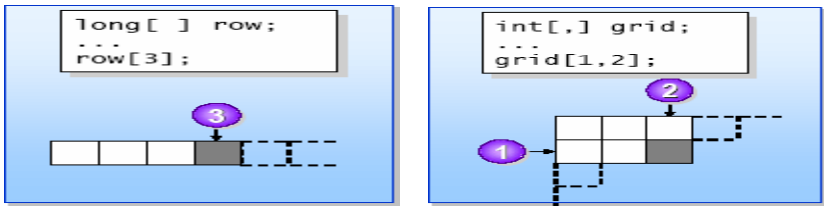

Figure 6 - Accès à des éléments de tableau à plusieurs dimensions

6.1.5. Vérification des limites de tableau

En C#, une expression d'accès à un élément de tableau est automatiquement vérifiée pour garantir que l'index est valide. Ce contrôle de limites implicite ne peut pas être désactivé. Le contrôle de limites est l'un des moyens permettant de garantir la sécurité du langage C#.

Pour vérifiez que les index entiers sont toujours à l'intérieur des limites, vérifiez manuellement les limites d'index, à l'aide d'une condition de fin d'instruction ***for***, comme suit :

```
for (int i = 0; i < row.Length; i++) {
   Console.WriteLine(row[i]);
}
```

La propriété ***Length*** est égale à la longueur totale du tableau, quel que soit le rang du tableau. Pour déterminer la longueur d'une dimension spécifique, vous pouvez utiliser la méthode ***GetLength***, comme suit :

```
for (int r = 0; r < grid.GetLength(0); r++) {
   for (int c = 0; c < grid.GetLength(1); c++) {
      Console.WriteLine(grid[r,c]);
   }
}
```

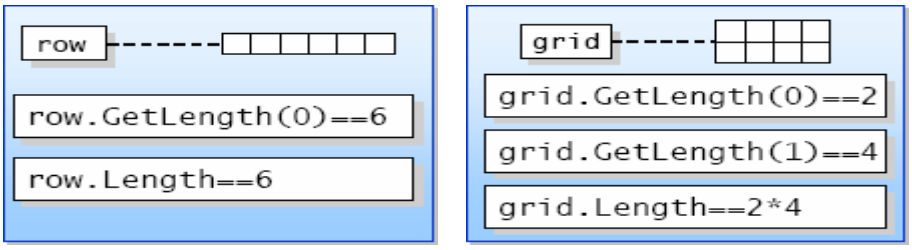

Figure 7 - Propriétés Length et GetLength

6.2. Création de tableaux

6.2.1. Création d'instances de tableau

Vous utilisez le mot clé ***new*** pour créer une instance de tableau, également appelée expression de création de tableau. Vous devez spécifier la taille de toutes les longueurs de rang lorsque vous créez une instance de tableau.
Le compilateur C# initialise implicitement chaque élément du tableau à une valeur par défaut qui varie en fonction du type de l'élément : les entiers sont implicitement initialisés à 0, les éléments à virgule flottante à 0.0 et les éléments booléens à ***false***.

```
long[ ] row = new long[4];
```

6.2.2. Initialisation d'éléments de tableau

Vous pouvez utiliser un initialiseur de tableau pour initialiser les valeurs des éléments de l'instance du tableau. Un initialiseur de tableau est une séquence d'expressions entre accolades et séparées par des virgules. Les initialiseurs de tableau sont exécutés de gauche à droite et peuvent comprendre des appels de méthode et des expressions complexes, comme dans l'exemple ci-dessous :

```
int[ ] data = new int[4]{1, 2, 3, 4};
```

6.2.3. Initialisation d'éléments de tableaux multidimensionnels

Vous devez initialiser explicitement tous les éléments du tableau, quelle que soit la dimension du tableau :

```
int[,] data = new int[2,3] { {42, 42, 42},
                             {42, 42, 42} };
```

7. Notions fondamentales de la programmation orientée objet

C# est un langage orientée objet. Tout développeur C# (et tout développeur .NET) se doit de maîtriser les termes et les principes de base de la programmation objet. Après avoir abordé les termes et les concepts généraux de la programmation objet, nous verrons son implémentation avec C#.

7.1. Classes et objets

La programmation orienté objet possède son propre vocabulaire et ses propres concepts que nous devons définir et assimiler avant de présenter plus concrètement son implémentation dans C#.

7.2. Qu'est-ce qu'une classe ?

Une classe est la définition d'un ensemble d'entités qui partagent les mêmes attributs, les mêmes opérations et la même sémantique.
Du point de vue du développeur, une classe est une construction syntaxique nommée qui décrit un comportement et des attributs communs

Exemple :
Toutes les voitures ont un comportement commun (elles tournent, s'arrêtent, etc.) et des attributs communs (quatre roues, un moteur, etc.). Vous employez le mot voiture pour faire référence à ces propriétés et ces comportements communs. Imaginez que l'on ne puisse pas les classer en concepts nommés : au lieu de dire voiture, il faudrait énumérer toutes les significations du mot voiture.

7.3. Qu'est qu'un objet ?

Un objet est une instance d'une classe et est caractérisé par une identité, un état et un comportement.

- Identité : les objets sont reconnaissables les uns des autres.

L'identité permet de distinguer un objet parmi tous les autres au sein d'une même classe. Si nous définissons une classe ***voiture***, votre voiture personnelle possédant un numéro de série particulier est un objet de la classe ***voiture***.

- Etat : les objets stockent des informations.

L'état d'un objet est une image des valeurs des attributs de l'objet à un instant donné. Il est possible que deux objets possèdent le même état, il n'en perde pas moins leur identité. Nous verrons plus tard que l'état d'un objet ne doit pas être accessible (principe d'encapsulation).

- Comportement : les objets réalisent des tâches

Le comportement d'un objet est sa finalité. Il peut effectuer des actions sur ses propres attributs ou sur ceux d'un tiers objet afin de basculer d'un état A à un état B.

Exemple :
Une personne a un nom, une date de naissance, une adresse et peut marcher, courir, sauter etc....
Jean Martin, né le 23 décembre 1984 est une instance de la classe personne.

7.4. Comparaison entre une classe et une structure

Une structure est un modèle de valeur alors qu'une classe est un modèle d'objets.

Une structure n'a pas d'identité et deux structures ayant le même état fonctionneront de la même manière que vous utilisiez l'une ou l'autre. L'état d'une structure est directement accessible et ne possède pas forcément de comportement.
Les structures peuvent effectuer des opérations, mais cela n'est pas recommandé. Il est préférable qu'elles ne contiennent que des données.

Comme nous l'avons vu précédemment, une classe possède une identité, un état inaccessible et un comportement. Les types représentés par des classes sont appelés des types référence en C#. Par rapport aux structures, rien à l'exception des méthodes ne doit être visible dans une classe bien conçue.

Types valeur et Types référence : Une valeur de type valeur est présente dans la pile de l'application en tant que variable locale ou en tant qu'attribut de l'objet auquel elle appartient.
Une valeur de type référence ne contient qu'un pointeur vers ses données présentes dans le tas managé.

7.5.Utilisation de l'encapsulation

L'encapsulation est le mécanisme permettant de rassembler les attributs et méthodes propres à un type donné afin d'en restreindre l'accès et/ou d'en faciliter l'utilisation et la maintenance.

L'encapsulation permet de masquer à l'utilisateur d'une classe son implémentation, c'est-à-dire les données qu'elle possède et les opérations qu'elle effectue afin de ne dévoiler que ce qui lui est réellement utile.

Exemple :
Lorsque vous effectuez un virement bancaire, un certain nombre de transactions et vérifications sont effectuées par votre banque. Ces opérations ne vous sont pas dévoilées (imaginez que vous puissez directement modifier le solde de votre compte…). Vous ne pouvez et n'avez pas à effectuer vous-mêmes ces opérations. De plus, si ces opérations sont mises à jour ou qu'une nouvelle vérification est effectuée, vous ne modifierez pas votre manière d'effectuer le virement.

De manière générale, les données sont privées et les méthodes sont publiques. Cependant certaines méthodes peuvent être privées et ne servir qu'au fonctionnement interne de la classe.

7.6.Données de l'objet, données statiques et méthodes statiques

Chaque objet possède ses propres données. Il est possible que deux objets distincts aient des valeurs semblables pour un même attribut mais il ne s'agît alors que d'une coïncidence.

On peut toutefois avoir besoin de définir une valeur commune pour l'attribut d'un type. L'utilisation de données statiques permet de partager une même valeur d'un attribut d'une classe avec tous les objets de cette classe.

Exemple :
Lors de la définition d'un compte épargne, le taux d'intérêt de ce type de compte sera commun à tous les comptes ouverts.

Ces données statiques peuvent être amenée à être modifié. On utilise alors des méthodes statiques appelées directement sur la classe. Les méthodes statiques ne peuvent agir que sur les données statiques.

7.7.C# et l'orientation objet

Voyons dès à présent comment C# utilise ses concepts.

7.7.1. Définition de classes simples

class BankAccount {	Une classe est définie à l'aide du mot clé ***class*** qui précède le nom de la classe.

private decimal balance; private string name; … public void Withdraw(decimal amount)) { ... } public void Deposit(decimal amount) { ... } }	Les données sont définies à l'intérieur de la classe comme nous le ferions avec une structure en veillant tout de même à rajouter le mot clé ***private*** afin de rendre celles-ci privées. Les opérations que peut réaliser la classe sont représentés par des méthodes que nous ferons précéder par le mot clé ***public*** s'il s'agît d'un comportement que nous voulons rendre accessible ou par ***private*** si l'opération n'est destinée qu'à être effectuée par l'objet lui-même.

Si l'accès à un attribut de classe n'est pas défini, le compilateur C# le considèrera par défaut comme privé.

7.7.2. Instanciation de nouveaux objets

La création d'une classe constitue la définition d'un type particulier. La création d'un objet appartenant à ce type se nomme l'instanciation.

Chaque objet doit être déclaré.
Afin d'instancier un nouvel objet déclaré, nous utiliserons le mot-clé ***new***.

```
class Program
{
	static void Main( )
	{
		BankAccount yours = new BankAccount( );
		yours.Deposit(999999M);
	}
}
```

7.7.3. Utilisation du mot clé this

Le mot clé ***this*** fait référence (peut être considéré comme le seul pointeur utilisé en C#) à l'objet à partir duquel il est appelé.

Ainsi, dans la classe ci-dessous les deux lignes de code sont équivalentes.

```
class BankAccount
{
	this.name = name;
	name = name;
}
```

Le mot clé ***this*** est très utilisé lorsque un conflit de nom apparaît.

```
class BankAccount
{
	public void SetName(string name)
	{
		this.name = name;
	}
}
```

L'utilisation du mot clé ***this*** permet ici d'assigner la valeur du paramètre ***name*** passée à la méthode ***SetName*** à l'attribut ***name*** de l'objet.

Attention : en cas de conflit de nom, aucun avertissement n'est indiqué par compilateur C# et l'attribut ***name*** de l'objet ne sera pas modifié.

Le mot clé ***this***, faisant appel à l'objet et non à la classe elle-même, vous ne pouvez pas l'utiliser afin d'accéder aux données et aux méthodes statiques.

7.7.4. Classes imbriquées

Il est possible de créer des classes à l'intérieur d'une autre classe, on parle alors de classes imbriquées.

Une classe imbriquée se déclare au sein même de la classe et peut être publique ou privée.

```
class Bank
{
      public class Account { .... }
      private class AccountNumberGenerator { .... }
}
```

7.8. Définition de systèmes orientés objet

Dans cette partie, nous traiterons de certains mécanismes propres à la programmation orientée objet tel que l'héritage ou le polymorphisme.

7.8.1. Héritage

L'héritage est le mécanisme au moyen duquel les éléments les plus spécifiques incorporent la structure et le comportement d'éléments plus généraux.
Une classe qui hérite d'un type devient en quelque sorte, une précision ou une spécialisation de ce type.
La classe la plus générale est appelé classe de base ou classe parente, la plus spécifique, la classe dérivée.

Exemple :
Un violoniste est un type, un musicien en est un autre. Cependant, un violoniste est également et avant tout un musicien. On peut donc faire hériter le type violoniste des caractéristiques de tout musicien.
Toute modification apportée au type musicien sera ainsi directement reportée au type violoniste.

7.8.2. Hiérarchie des classes

Un type dérivé peut être dérivé à son tour.
L'ensemble de ces classes forme alors une hiérarchie des classes dont on peut énumérer les niveaux.

Lorsque l'on remonte dans la hiérarchie, nous parlons de généralisation, à l'inverse, la descente niveau par niveau de la hiérarchie des classes représente une spécialisation.

Il est essentiel de limiter le nombre de niveaux d'une hiérarchie de classes afin d'en garder le contrôle. Une hiérarchie des classes doit normalement se limiter à 6 ou 7 niveaux.

7.8.3. Héritage simple et multiple

Lorsqu'une classe dérive d'une seule et même classe, on parle d'héritage simple. L'héritage multiple consiste à faire dériver une classe de plusieurs classes.

Exemple :

Une voiture est un véhicule et peut donc hériter des caractéristiques communes à tout véhicule (rouler, tracter etc.).
Une voiture peut également être un objet de collection et hériter caractéristiques communes des objets de collections (côte, nombre d'exemplaires etc.).
De ce point de vue, il vous incombe de choisir ce que vous souhaitez faire de votre voiture. Si vous souhaitez voyager avec votre voiture, vous l'utiliserez comme un objet de type véhicule. Si vous voulez la revendre, vous l'utiliserez peut être comme un objet de collection.

L'héritage multiple vous amène à faire un très mauvais usage de celui-ci. C'est pourquoi C# n'autorise pas ce procédé. Vous pouvez n'hériter que d'un seul type. Pour palier à l'absence d'héritage multiple, nous verrons dans la présentation des interfaces que nous pourrons implémenter plusieurs interfaces dans une même classe.

7.8.4. Polymorphisme

Le polymorphisme est le mécanisme permettant à un objet de prendre la forme de plusieurs types en fonction des types de bases desquels ils héritent ou des interfaces qu'il implémente.

L'exemple ci-dessus illustrant l'héritage multiple peut être repris pour illustrer le polymorphisme. En effet, une voiture peut à un instant t est utilisé comme

7.8.5. Classes de base abstraites

Certaines classes existent dans le seul but d'être dérivée, on parle de classe abstraite.
Les classes abstraites ne peuvent être instanciées.

Les classes abstraites contiennent très souvent des méthodes statiques afin de pouvoir agir sur des objets de type dérivé de la classe abstraite.

7.8.6. Interfaces

Une interface représente un contrat que la classe qui l'implémente se doit de remplir.
Une interface ne contient que des opérations (méthodes non implémentées) et ne peut donc pas être instanciée.

Exemple :
Un ordinateur et un téléviseur sont deux types qui peuvent être allumés. Cependant, il est difficilement concevable de faire hériter l'un de l'autre. C'est pourquoi, il conviendra ici de déclarer une interface contenant l'opération allumer et de l'implémenter dans les classes ordinateur et téléviseur.

7.8.7. Liaison anticipée et tardive

La liaison anticipée (ou liaison statique) est le fait d'effectuer un appel de méthode directement sur l'objet.

La liaison tardive (ou liaison dynamique) se présente lorsque que l'on effectue l'appel de méthode non plus sur l'objet lui-même mais via une opération de classe de base.

8. Utilisation de variables de type référence

Cette partie concerne l'étude des variables de types références en C#, de la hiérarchie des classes et plus particulièrement de la classe de base ***System.Object***, des différentes conversions entre ces types et des conversions ***boxing/unboxing.***

8.1. Comparaison entre les types valeur et les types référence

Les variables de type valeur, les plus simples, (***int***, ***long*** ou ***char***), contiennent leur valeur directement dans la mémoire (pile).

Les données de type référence contiennent une référence aux données stockées dans une zone séparée de la mémoire (***tas managé***).

On peut comparer les variables de type référence aux pointeurs en C et C++ sans ne jamais omettre qu'en C#, une variable de type référence ne peut jamais désigner un objet invalide. De plus, aucune gestion de la mémoire n'est nécessaire puisque cela est strictement réservé au ***Garbage Collector***.

8.2. Déclaration et libération des variables référence

La déclaration d'une variable de type référence ne diffère pas de la déclaration de variable de type valeur.

Prenons pour exemple la classe ***Coordinate*** représentant les coordonnées d'un point.

```
class coordinate
{
      public int x;
      public int y;
}
```

On pourra déclarer des coordonnées de la façon suivante :

```
coordinate c1;
```

Cependant, je n'ai ici fait que déclarer la référence vers un ***Coordinate*** et il m'est donc impossible d'assigner des valeurs à mon objet.
Pour cela, je dois désormais utiliser l'opérateur ***new*** afin de créer les données que référencera c1 :

```
c1 = new coordinate();
```

Lors de l'instanciation, un appel vers le constructeur de l'objet est effectué. Cependant, ma classe ***Coordinate*** n'implémente aucun constructeur. Le compilateur C# appelle donc un constructeur implicite (constructeur par défaut) qui va initialiser les attributs de type simple (type valeur) à leur valeur d'origine. Actuellement, nous aurons donc :

```
c1.x = 0;
c1.y = 0;
```

Cependant, si aucun constructeur n'est implémenté, les objets de type référence déclaré dans la classe ne peuvent être instanciés.

En C#, vous ne pouvez accéder à des références invalides. Lors de la compilation, une erreur empêche la compilation si une référence n'est pas initialisée et une exception est levée lors d'une tentative d'accès à une référence non valide pendant l'exécution.

Ainsi, si nous déclarons un objet de type ***System.Random***, qui permet de générer des nombres aléatoires dans notre classe ***Coordinate***, nous ne pourrons pas l'utiliser et à l'exécution, une exception de type ***System.NullReferenceException*** sera levée.

```
class Coordinate
{
	public System.Random rnd;
	public int x;
	public int y;
}
```

```
…
c1.rnd.Next();  //exception levée!!!
…
```

Afin d'accéder aux variables membres, C# utilise la notation pointée

```
c1.x = 6.12;
c1.y = 4.2;
```

c1 fait désormais référence à un objet de type ***Coordinate*** dont les attributs x et y ont été respectivement assignées à 6,12 et 4,2.

Pour libérer c1, on utilise le mot clé ***null***.

```
c1.x = null;
```

8.3.Comparaison de valeurs et comparaison de références

Un autre exemple montre comment c1 est une référence et non pas l'objet lui-même. Créons désormais deux objets ***Coordinate*** dont les valeurs x et y sont assignées aux mêmes valeurs et comparons les.

```
class Program
{
	static void Main(string[] args)
	{
		Coordinate c1 = new Coordinate();
		Coordinate c2 = new Coordinate();

		c1.x = 1.0;
		c1.y = 2.0;

		c2.x = 1.0;
		c2.y = 2.0;

		if(c1 == c2)
			Console.WriteLine("Même chose!!!");
		else
			Console.WriteLine("Pas la même chose!!!");
		Console.Read();
	}
}
```

La sortie est la suivante :

```
Pas la même chose!!!
```

En effet même si ***c1*** et ***c2*** ont les mêmes valeurs, ils font référence à deux objets différents.

Pour que deux variables de type référence soient égales, il faut qu'il fasse référence au même objet.

```
class Program
{
      static void Main(string[] args)
      {
            Coordinate c1 = new Coordinate();
            Coordinate c2;

            c2 = c1;  //c2 référence le même objet que c1

            if(c1 == c2)
                  Console.WriteLine("Même chose!!!");
            else
                 Console.WriteLine("Pas la même chose!!!");
            Console.Read();
      }
}
```

```
Même chose!!!
```

Les deux variables sont désormais équivalentes puisque faisant référence au même objet.

8.4. Utilisation de références comme paramètres de méthode

Les variables de type référence peuvent être utilisées comme paramètres de méthodes. Nous allons ici présenter la différence entre l'utilisation de type référence et de type valeur comme paramètre de méthodes.

Nous avons vu auparavant que les structures étaient des variables de type valeur. Transformons donc la classe ***Coordinate*** en une structure ***StructCoordinate***.

```
struct StructCoordinate
{
      public int x;
      public int y;
}
```

Ecrivons maintenant une classe ***Program*** implémentant une méthode incrémentant les coordonnées. Cette méthode sera surchargée afin de pouvoir accueillir une référence ou une valeur comme paramètre.

```
class Program
{
	static void Main(string[] args)
	{
		Coordinate  c1 = new Coordinate();
		StructCoordinate  c2 = new StructCoordinate();

		//Modification des coordonnées
		ChangeCoordinate(c1);
		ChangeCoordinate(c2);

		Console.WriteLine(string.Format("c1: {0};{1}",c1.x,c1.y));
		Console.WriteLine(string.Format("c2: {0};{1}",c2.x,c2.y));

		Console.Read();
	}

	//Méthode acceptant une référence comme paramètre
	static void ChangeCoordinate(Coordinate coor)
	{
		coor.x++;
		coor.y++;
	}

	//Méthode acceptant une valeur comme paramètre
	static void ChangeCoordinate(StructCoordinate coor)
	{
		coor.x++;
		coor.y++;
	}
}
```

Sortie console :

```
C1 : 1 ;1
C2 : 0 ;0
```

L'action de la méthode ***ChangeCoordinate*** est différente suivant le type de variable transmise comme paramètre.
Lorsque vous fournissez des variables de type valeur, la méthode agît sur une copie de la structure de données. Dans le cas présent, c'est une copie de la structure ***c2*** qui est transmise à la méthode. Cette copie est une variable locale et sera détruite à la fin de la méthode.

L'utilisation d'une référence comme paramètre fournit à la méthode l'emplacement des données dans le ***tas managé***. Ainsi, les modifications apportées aux données s'effectuent sur les données référencées par ***c1*** et sont donc conservées après l'exécution de la méthode.

On peut outrepasser le fait que le paramètre utilisé dans une méthode soit une copie de la variable passée en paramètre à l'aide de l'utilisation du mot clé ***ref***.

On peut modifier le code précédent pour passer la structure en ajoutant le mot clé ***ref***.

```
static void Main(string[] args)
{
      StructCoordinate c1 = new StructCoordinate();

      ChangeCoordinate(ref c1);

      Console.WriteLine(string.Format("c1 : {0};{1}",c1.x,c1.y));
      Console.Read();
}

static void ChangeCoordinate(ref StructCoordinate coor)
      {
            coor.x++;
            coor.y++;
}
```

Sortie console :

```
C1 : 1 ;1
```

Les modifications apportées ont cette fois-ci était conservées après l'exécution de la méthode.

Pour utiliser le mot clé ***ref***, nous devons instancier l'objet avant de le passer en paramètre à la méthode. Le mot clé ***out*** permet d'instancier l'objet passé en paramètre seulement au niveau de la méthode appelée.

```
static void Main(string[] args)
{
      StructCoordinate c1;  // //l'objet peut ne pas être instancié

      ChangeCoordinate(out c1);

      Console.WriteLine(string.Format("c1 : {0};{1}",c1.x,c1.y));
      Console.Read();
}

static void ChangeCoordinate(out StructCoordinate coor)
      {
            //l'objet peut et doit être instancié dans la méthode
            c1 = new StructCoordinate();

            coor.x++;
            coor.y++;
}
```

On obtient alors le même résultat qu'avec l'utilisation de ***ref***. Seul l'endroit d'instanciation diffère.

8.5.Utilisation de type référence courants

Cette partie a pour objectif de vous familiariser avec certains types références faisant partie intégrante du langage C#.

8.5.1. System.Exception

La classe ***System.Exception*** dont nous avons déjà abordé l'utilisation est un type référence. Cette classe représente une erreur générique dans une application. Les erreurs peuvent être déclenchées à l'aide de l'opérateur ***throw***.
De nombreuses classes dérivées de ***System.Exception*** afin de mieux cibler l'origine de l'erreur :

System.NullReferenceException
System.IO.FileNotFoundExecption
System.NET.ProtocolViolationException
…

Vous pouvez de plus créer vos propres exceptions en créant une classe héritant de ***System.Exception***.

8.5.2. System.String

8.6. Hiérarchie des objets

Toutes les classes en C# héritent directement ou indirectement du type ***System.Object***. Lorsque vous créez une classe sans préciser d'héritage, celle-ci hérite directement de ***System.Object***. Cette partie décrira donc quelques méthodes essentielles de cette classe que vous pourrez retrouver dans toutes les classes.

8.6.1. Méthode ToString

La méthode ***ToString*** retourne une chaîne représentant l'objet en cours. Par défaut, son implémentation retourne le nom de la classe.

```
Console.WriteLine(new object().ToString());
```

Cette ligne affichera : ***System.Object***

Cependant, cette méthode peut être redéfinie selon l'usage qu'on fait de la classe.
Ainsi, pour un ***int*** il est plus efficace de retourner la valeur contenu dans la variable sous forme de chaîne.

```
int i = 3;
Console.WriteLine(i.ToString());
```

La dernière ligne affichera: 3

8.6.2. Méthode Equals

La méthode ***Equals*** permet de comparer deux objets.

```
…
Coordinate a = new Coordinate();
Coordinate b = new Coordinate();

a.Equals(b);            //renvoit False

a = b;

a.Equals(b);            //renvoit True
…
```

Cependant certains types redéfinissent cette méthode afin de comparer les valeurs contenues comme c'est le cas pour la classe ***string***.

```
string a = "salut";
string b = "salut";

a.Equals(b);            //renvoit True
```

8.6.3. Méthode GetType

La méthode ***GetType*** retourne des informations d'exécution de l'objet, nous traiterons plus amplement de cette méthode lors de l'étude de la réflexion.

8.6.4. Méthode Finalize

La méthode ***Finalize*** est appelée automatiquement par le ***runtime*** lorsque l'objet devient inaccessible (si celui-ci n'est plus référencé).

8.6.5. Méthode GetType

La méthode ***GetType***, retourne un objet de type ***System.Type*** incluant de méthodes permettant d'obtenir des informations sur les constructeurs, méthodes, les champs et les propriétés implémentés dans la classe.

8.6.6. Opérateur TypeOf

L'opérateur ***TypeOf*** permet de récupérer l'objet ***System.Type*** de l'objet passé en paramètre. L'opérateur ***TypeOf*** ne fonctionne qu'avec des types connus au moment de la compilation.

8.6.7. Réflexion

La réflexion vous permet de récupérer dynamiquement (à l'exécution), des informations sur les types utilisés. Les classes utiles à la réflexion sont regroupées dans l'espace de noms ***System.Reflexion***. Ces classes sont nombreuses et une étude exhaustive de la réflexion sort du cadre de ce cours.

8.7.Espace de noms du Framework.NET

Les espaces de noms (namespaces) permettent d'organiser le code et de définir des portées de noms.

Les noms globaux sont composés des noms des différents espaces de noms suivis du nom de type.
Le type ***System.Web.Mail.MailMessage*** par exemple est contenu dans le namespace ***System.Web.Mail.***
Si vous souhaitez écrire votre propre classe ***MailMessage***, vous pouvez inclure celle-ci dans votre propre espace de noms à l'aide de l'instruction ***namespace{ }***.

```
namespace LaboDotNet.Cours.Methodes
{
      class Read
      {
            ...
      }

      class Write
      {
            ...
      }
}
```

En général, on utilise le prototype suivant :

<nom de l'entreprise>.<nom de l'application>.<nom de la bibliothèque>.<…>.<nom du type>

Tous les types en C# sont inclus dans un espace de noms, même si vous n'en précisez pas (par défaut les types sont contenus dans un espace de noms portant le nom de l'application).

Les espaces de noms permettent donc de conserver des noms de type explicites en évitant les conflits de noms tout en organisant le code.

Cependant, les noms globaux ne sont pas pratiques à manipuler et l'on préférerait s'en tenir au nom de type dans le code.
La directive ***using*** permet de spécifier les différents espaces de noms avec lesquels on souhaite travailler dans le fichier en cours.

```
using System;
using System.IO;
using SQL = System.Data.SqlClient;

namespace Methodes_Object
…
```

Dans l'exemple ci-dessus, un alias a été défini pour l'espace de noms ***System.Data.SqlClient***.

Passons désormais à la présentation de quelques espaces de noms utiles à l'aide de C#.

8.7.1. Espace de noms System.IO

L'espace de noms ***System.IO*** contient entre autre l'ensemble des classes du Framework.NET destinées à gérer les entrées-sorties vers le système de fichiers.

Classes ***File*** et ***Directory***

File et ***Directory*** sont des classes abstraites implémentant des méthodes statiques vous permettant de créer, supprimer et manipuler les répertoires et les fichiers.

Classes ***StreamReader*** et ***StreamWriter***

StreamReader et ***StreamWriter*** permettent respectivemet d'accéder et d'écrire le contenu d'un fichier sous forme de flux d'octets ou de caractères.

Classes ***FileStream***

La classe **FileStream** peut être utilisée pour permettre l'accès aléatoire aux fichiers.

Classes ***BinaryReader*** et ***BinaryWriter***

BianryWriter et ***BinaryReader*** permettent de lire et d'écrire des types de données primitifs en tant que valeurs binaires.

8.7.2. Espace de noms System.Xml

System.Xml contient les classes vous permettant d'écrire des applications pouvant interagir avec des fichiers et flux au format XML.

L'étude du support d'XML dans .NET fait l'objet d'un cours complet.

8.7.3. Espace de noms System.Data

System.Data est l'espace de noms regroupant les classes nécessaires à la gestion des données. Cet espace de noms contient entre autre chose les espaces de noms et les classes permettant la connexion à différents SGBD.

System.Data est plus amplement étudié dans le cours d'ADO.NET.

8.8.Conversion de données

.NET est fortement typé et il est souvent nécessaire de convertir les données afin de remplir d'effectuer certaines opérations.
Cette section traitera donc de la conversion des types valeurs et références ainsi que du mécanisme de boxing/unboxing.

8.8.1. Conversions implicites

De nombreuses conversions se produites sans l'ajout d'instruction supplémentaires. Le compilateur effectue seul ces conversions car celles-ci ne peuvent entraîne de pertes d'informations.

```
int a = 6;
long b = a;       //conversion implicite de a en 'long'
```

8.8.2. Conversions explicites

Certaines conversions peuvent ne pas être sure et il incombera alors au développeur d'effectuer lui-même sa conversion. Par exemple la conversion d'un long en int.

```
long a = 6;
int b = a;

//Erreur : Impossible de convertir implicitement le type long en int
```

Nous pouvons donc forcer le compilateur à exécuter la conversion :

```
long a = 6;
int b = (int)a;
```

Cependant si ***a*** est incrémenté et que sa longueur dépasse la capacité d'un ***int*** (4 octets), la conversion sera refusée et une exception de type ***System. OverflowException*** sera levée.

8.8.3. Conversions de types référence

On peut distinguer deux types de conversions de types références, les conversions d'un type enfant vers un type parent et inversement d'un type parent vers un type enfant.

La conversion d'un type enfant vers un type parent peut être implicite, est sûre et toujours autorisé.

```
string a = "salut";
object o =  a;
```

La conversion d'un type parent vers un type enfant doit être explicitement effectuée. Aucune vérification n'est effectuée avant l'exécution de l'application. Lors de la conversion, les types sont vérifiés. Si une perte de données peut être causées par l'opération, elle sera annulée et une exception de type ***System.InvalidCastException*** sera levée.

L'opérateur ***is*** permet de vérifier le type d'un objet et renvoie ***True*** si les deux opérandes sont de mêmes types.

```
Bird b;
if(a is Bird)
{
      b = (Bird)a;
}
```

L'opérateur ***as*** permet d'effectuer une opération de conversion sans générer d'erreur en cas de cast invalide. Si la conversion n'est pas sûre, ***as*** renvoie ***null***.

8.8.4. Boxing/Unboxing

C# inclut un système de types unifiés qui permet de convertir les types valeur en références de type ***System.O*bject**, ainsi que de convertir des références d'objets en types valeur. Les types valeur peuvent être convertis en références de type ***System.O*bject**, et vice-versa.

Le ***Boxing*** consiste donc à convertir une variable de type valeur en un objet de type référence ***System.Object***.

```
int i = 123;
object box;

box = p;                //boxing implicite
box = (object)p;        //boxing explicite
```

L'***Unboxing*** consiste à convertir un type ***System.Object*** en un type valeur. Cette conversion ne peut être qu'explicite.

```
i = (int)box;
```

9. Création et destruction d'objets

Cette partie traite des mécanismes de création et de destruction des objets. Nous aborderons l'utilisation des constructeurs et des destructeurs et décrirons la durée de vie d'un objet.

9.1. Utilisation de constructeurs

En C#, la seule manière de créer un objet consiste à utiliser le mot clé ***new*** pour allouer la mémoire.

Il n'existe aucune exception à cette règle bien que le compilateur le fasse implicitement à votre place. Les deux lignes ci-dessous sont par exemple identiques :

```
string a = "salut";
string b = new string(new char[]{'s','a','l','u','t'});
```

La deuxième étape de création des objets consiste à appeler un constructeur. L'appel du constructeur crée l'objet dans l'espace mémoire alloué. On peut différencier deux types de constructeurs différents, les constructeurs d'instances ou d'objets et les constructeurs statiques ou constructeurs de classes.

9.1.1. Constructeurs d'instances

Après avoir fait usage du mot clé ***new***, un espace mémoire suffisant a été alloué, l'appel du constructeur va terminer la construction de l'objet en initialisant cet espace mémoire.

En C#, vous ne pouvez pas distinguer l'allocation de l'initialisation de la mémoire.
Si vous n'implémentez pas de constructeur dans votre classe, le compilateur fera appel à un constructeur par défaut. Celui-ci initialisera les variables de type valeur membres à leur valeur par défaut (ex : ***int*** = 0) mais ne construira pas les variables de type référence membres.

Un constructeur par défaut doit porter le même nom que la classe, être publique, ne prendre aucun argument et ne renvoyait aucune valeur (pas même ***void***).

Vous pouvez réécrire votre propre constructeur par défaut en suivant les règles présentées ci-dessus. La réécriture peut trouver son utilité si vous n'êtes pas satisfait des valeurs par défaut que prennent les membres de votre classe.

Dans l'exemple ci-dessous, le constructeur de la classe ***MyDate*** a était modifié afin que le membre ***DateTime dt*** soit initialisé aux dates et heures actuelles plutôt qu'au 01/01/01.

```
class MyDate
{
	public DateTime dt;

	public MyDate()
	{
		dt = DateTime.Now;
	}
}
```

Le constructeur peut être surchargé afin de passer des arguments différents et de pouvoir modifier l'initialisation des membres selon le type de données lors de la construction de l'objet.

```
class MyDate
{
	public DateTime dt;

	public MyDate(DateTime dt)
	{
		this.dt = dt;
	}

	public MyDate(int year,int month, int day)
	{
		dt = new DateTime(year,month,day);
	}

}
```

Attention : si vous surchargez le constructeur d'une classe, l'appel au constructeur par défaut implicite est impossible. Il vous faudra alors écrire votre propre constructeur par défaut.

Une liste d'initialiseurs peut vous permettre d'écrire un constructeur appelant un autre constructeur dans la même classe. Une liste d'initialiseurs commence par le signe deux-points, est suivie du mot clé **this**, puis d'arguments entre parenthèses.

```
class MyDate
{
	public DateTime dt;
	public MyDate(DateTime dt) : this(2004,01,01)
	{
		this.dt = dt;
	}

	public MyDate(int year,int month, int day)
	{
		dt = new DateTime(year,month,day);
	}
}
```

9.1.2. Constructeurs statiques

Les constructeurs statiques ont pour rôle de permettre l'initialisation des membres statiques de la classe.

```
class Tailor
{
	public static string helpMsg;
	static Tailor()
	{
		def = "My tailor is rich!!!";
	}
}
```

Vous ne pouvez appeler vous-mêmes un constructeur statique. Il est donc impossible de préciser l'accès au constructeur de classe ou d'ajouter des arguments.
Les constructeurs statiques sont aussi appelés constructeurs de classe car ils n'ont accès qu'aux membres statiques de classe. Ainsi, l'utilisation de ***this*** (référençant un objet) dans un constructeur statique entraîne une erreur de compilation.

9.1.3. Constructeurs de structures

Les membres d'une structure peuvent être initialisés à l'aide d'un constructeur de structure.

```
struct MaStruct
{
      public int x;
      public int y;

      public MaStruct(int a,int b)
      {
            x = a;
            y = b;
      }
}
```

Cependant, l'implémentation d'un constructeur de structures est soumise à certaines règles.

Tous les membres de la structure doivent être explicitement initialisés dans le constructeur.

```
public MaStruct(int a,int b)
{
      x = a;            //Erreur : Le champ y doit être totalement assigné
                        // avant que le contrôle quitte le constructeur.
}
```

Vous ne pouvez pas déclarer de constructeur par défaut

```
public MaStruct()
{
      x = 0;      //Erreur : Les structures ne peuvent pas contenir de
      y = 1;      // constructeurs  exempts de paramètres explicites
}
```

9.2.Objets et mémoire

En C# et en .NET, la gestion de la mémoire incombe entièrement et perpétuellement à la CLR. Nous allons décrire cette gestion et le fonctionnement du ***garbage collector***.

Nous avons vu comment initialiser des variables de type référence. Ce processus est composé des deux étapes que sont la réservation de l'espace mémoire et l'initialisation des variables. La destruction de l'objet est le processus exactement inverse, consistant à déinitialiser l'objet et à libérer la mémoire.

Cependant, le contrôle de ce processus ne nous vous est pas attribué. En effet, cette tâche est allouée au ***garbage collector***.
L'opération de destruction et son mauvais usage étant d'en d'autres langages une source importante d'erreurs grave, .NET automatise la tâche.
Le garbage collector a pour but de :

- Déterminer les objets inaccessibles (non référencés).
- Libération de la mémoire.

Cette tâche s'accomplit lorsque la mémoire devient insuffisante. Vous ne pouvez déterminer le moment de cette libération et donc de la destruction de vos objets.

9.3.Gestion des ressources

Si l'on ne peut déterminer l'instant où l'objet sera détruit, on peut implémenter certaines actions à effectuer lors de sa destruction.

9.3.1. Finalize

Les objets .NET héritent de ***System.Object*** de la méthode ***Finalize***. ***Finalize*** est exécutée lors de la libération de l'objet mais ne peut être ni appelée ni substituée.

9.3.2. Destructeurs

Un destructeur est à l'inverse d'un constructeur destiné à déinitialiser les membres d'un objet.

Un destructeur est une sorte de méthode de la classe, portant le même nom que celle-ci, ne possédant pas de modificateur d'accès, de type de retour (pas même ***void***), de paramètres. De plus, un constructeur est précédé par **~.**

```
class SourceFile()
{
      ~SourceFile()
      {
            ...
      }
}
```

Vous devez limiter l'utilisation des destructeurs car vous ne pouvez contrôler leur moment d'exécution. Cela pourrait entraîner des ralentissements dans le processus de libération de la mémoire et ainsi générer des pertes de performances.

9.3.3. IDisposable et Dispose

Vos propres classes peuvent être amener à utiliser des ressources dont vous souhaitez contrôler la libération. Le processus de libération des objets ne vous permettant pas libérer ces ressources à un instant donné, l'implémentation d'une méthode ***Dispose*** vous permet d'accomplir cette tâche.

10. Héritage dans C#

L'héritage en C# comme dans tout langage orienté objet, est la capacité qu'à une classe à hériter des données et des fonctionnalités d'un autre type.
Cette partie traite de l'héritage en C#, de la façon de dériver des classes à partir de classes existantes et des types de méthodes ***virtual***, ***override*** et ***new***.

10.1. Dérivation de classe

10.1.1. Syntaxe

La syntaxe pour faire hériter d'une classe de base est la suivante :

```
class Derived : Base
{
	...
}
```

Une classe dérivée ne peut pas être plus accessible que sa classe de base. Par exemple, il n'est pas possible de dériver une classe publique d'une classe privée. La syntaxe C# pour dériver une classe d'une autre classe est également autorisée en C++ où elle spécifie implicitement une relation d'héritage privé entre les classes dérivées et les classes de base. C# n'autorise pas l'héritage privé ; tous les héritages sont publics.

10.1.2. Utilisation du mot clé protected

Le mot clé ***protected*** permet de restreindre l'accès d'un membre d'une classe de base uniquement à ses classes dérivées. Le membre protégé hérité devient automatiquement protégé.
Pour les autres types (ne dérivant pas de la classe), un membre protégé d'une classe se comporte comme un membre privé

10.1.3. Appel de constructeurs de classe de base

Vous avez la possibilité de définir des constructeurs faisant appel à un constructeur de la classe de base afin d'initialiser les membres hérités de cette classe.

```
class Vehicule
{
	protected string immatriculation;

	public Vehicule(string immatriculation)
	{
		this.immatriculation = immatriculation;
	}
}

class Voiture : Vehicule
{
	public Voiture(string immatriculation) : base(immatriculation)
	{
		...
	}
}
```

10.2. Implémentation de méthodes

L'héritage offre des mécanismes de réutilisation du code et de redéfinition des méthodes. Ce mécanisme est appelé polymorphisme.

10.2.1. Utilisation de virtual et override

Une méthode virtuelle de la classe de base peut être redéfinie dans une classe dérivée.
Une classe virtuelle signifie que la méthode est une implémentation possible mais que l'on autorise sa redéfinition dans les classes dérivées.
Seuls les méthodes virtuelles non statiques et/ou privées peuvent être redéfinies dans une classe dérivée. Les deux méthodes doivent avoir la même signature, c'est-à-dire le même nom, le même modificateur d'accès, le même type de retour et les mêmes paramètres.
Pour déclarer une méthode virtuelle dans la classe de base, on utilise le mot clé ***virtual***. Dans la classe dérivée, on utilise le mot clé ***override*** pour redéfinir une méthode virtuelle de la classe de base.

```
class Vehicule
{
	public virtual void Deplacer()
	{
		...
	}

}

class Voiture : Vehicule
{
	public override void Deplacer()
	{
		...
	}
}
```

10.2.2. Utilisation de new

Plutôt que de redéfinir une méthode, vous pouvez avoir besoin de spécifier qu'une méthode d'une classe dérivée est une nouvelle implémentation de la méthode.
On utilise ***new*** pour spécifier une méthode dont la signature est équivalente à une méthode de la classe de base mais que nous souhaitons masquer. Le fait de masquer la méthode virtuelle de la classe de base permet également de résoudre des conflits de noms au sein des classes dérivées.

```
class Bateau :Vehicule
{
	...
	public new void Deplacer()
	{
		...
	}
}
```

10.3. Utilisation d'interfaces

Une interface est une définition syntaxique d'un ensemble d'opérations que les classes l'implémentant se doivent de respecter.
Ainsi, si vous connaissez les interfaces implémentées dans une classe, vous pouvez être certain que certaines méthodes de l'objet pourront être appelées.
Si une classe ne peut dérivée que d'une seule classe de base, les classes C# peuvent implémenter divers interfaces.

La définition d'une interface est similaire à la définition d'une classe sans implémentation. On utilise donc le mot clé interface et les méthodes déclarées dans l'interface ne doivent pas contenir de code.

```
interface IDeplacable
{
      void Deplacer();
}
```

Une interface ne comprend que des méthodes. Les membres sont interdits dans une interface et les méthodes de l'interface ne peuvent posséder de modificateurs d'accès.

La classe implémentant l'interface doit obligatoirement implémenter l'ensemble des méthodes de l'interface.

```
class Voiture : Vehicule,IDeplacable
{
      public void Deplacer()
      {

      }
}
```

Si la classe ***Voiture*** n'implémentait pas la méthode ***Deplacer***, une erreur serait survenue à la compilation.
Une utilisation possible des interfaces consiste à itérer les éléments d'une collection, dont on ignore le type exact.

```
ArrayList mesVehicules = new ArrayList();
mesVehicules.Add(new Voiture());
mesVehicules.Add(new Bateau());

foreach(IDeplacable i in mesVehicules)
{
      i.Deplacer();
}
```

Comme mes véhicules implémentent l'interface ***IDeplacable***, je suis certain que chaque véhicule implémente la méthode ***Deplacer***.

Exemple :

```
// interface.cs

using System;

interface IDimensions
{
      float Length();
      float Width();
}

class Box : IDimensions
{
      float lengthInches;
      float widthInches;

      public Box(float length, float width)
      {
```

```
            lengthInches = length;
            widthInches = width;
      }
      float IDimensions.Length()
      {
            return lengthInches;
      }
      float IDimensions.Width()
      {
            return widthInches;
      }

      public static void Main()
      {
            Box myBox = new Box(30.0f, 20.0f);
            IDimensions myDimensions = (IDimensions) myBox;
            System.Console.WriteLine("Length: {0}", myDimensions.Length());
            System.Console.WriteLine("Width: {0}", myDimensions.Width());

            Console.ReadLine();
      }
}
```

Sortie console :

```
Length: 30
Width: 20
```

Explications :

- Nous déclarons tout d'abord l'interface en précisant quels seront les membres et méthodes à implémenter. En effet, utiliser les interfaces permet de forcer le développeur à développer un certain nombre de méthode. Dans notre exemple, nous déclarons deux méthodes (***Length()*** et ***Width()***). Notre classe ***Box*** implémentant cette interface, le développeur doit alors développer ces méthodes. ***Box*** implémente ces interfaces en mettant en nom de méthode ***IDimensions.Length()*** par exemple, ce qui indique que c'est l'implémentation de ***Length*** de l'interface ***IDimensions***.
- On peux alors appeler la méthode d'une interface depuis un objet, et ce quel que soit son type, tant que l'objet implémente l'interface.
- Si besoin, le développeur peut implémenter autant d'interface qu'il le souhaite
- Commencez obligatoirement le nom de votre interface par un 'I'.
- Le Framework .NET comprend un grand nombre d'interface que vous pouvez utiliser. Par exemple, l'interface ***ICollection*** vous permet de créer des collections personnalisées.

10.4. Utilisation des classes abstraites et scellées

Certaines classes n'existent que dans le but d'être dérivées.
On utilise le mot ***abstract*** dans la déclaration de la classe afin de la déclarer abstraite.

```
abstract class Vehicule
{
      public virtual void Deplacer()
      {
            ...
      }
}
```

Une classe abstraite ne peut être instanciée.

Au contraire des classes abstraites, certaines classes peuvent être scellées. Les classes scellées ne peuvent être dérivées. Pour déclarer une classe scellée, on utilise le mot clef ***sealed.***

11. Opérateurs, délégués et évènements

11.1. Surcharge d'opérateurs

La surcharge d'opérateur permet de contrôler le comportement d'un objet face à un opérateur.

Saisissez et exécutez le programme suivant :

```
// OperatorOverload.cs

using System;

public class Complex
{
	public int Real;
	public int Imaginary;

	public Complex(int Real, int Imaginary)
	{
		this.Real = Real;
		this.Imaginary = Imaginary;
	}

	public static Complex operator +(Complex c1, Complex c2)
	{
		return new Complex( c1.Real + c2.Real,
				c1.Imaginary + c2.Imaginary);
	}

	public override string ToString()
	{
		return(String.Format("{0} + {1}i", Real, Imaginary));
	}
}

public class OperatorOverload
{
	static void Main()
	{
		Complex num1 = new Complex(2,3);
		Complex num2 = new Complex(3,4);
		Complex sum = num1 + num2;

		Console.WriteLine("First complex number:  {0}",num1);
		Console.WriteLine("Second complex number: {0}",num2);
		Console.WriteLine("The sum of the two numbers: {0}",sum);

		Console.ReadLine();
	}
}
```

Explications :

- Nous avons tout d'abord créé une classe ***Complex*** contenant deux membres publics.
- Nous ne pouvons additionner deux objets directement. Il faut donc surcharger l'opérateur +. Remarquez que la méthode est statique.

- Cette méthode retourne un ***Complex*** et prends deux arguments de type ***Complex***, mais nous pouvons très bien créer autant de méthode que nécessaire pour les autres types (avec des ***int*** en paramètres par exemple).
- Les opérateurs du c# sont : +, -, !, ~, ++, --, true, false, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=.

11.2. Délégations

Les délégations sont un type un peu particulier. Elles permettent de fournir une fonction en paramètre pour déléguer l'exécution d'un morceau de programme. C'est l'équivalent des pointeurs de fonctions en C/C++.

Exécutez ce code :

```
// Delegating.cs

using System;

public delegate void DelegateType(int i);

public class Delegating
{
	public static void Main()
	{
		DelegateType del=new DelegateType(MyFunction);
		del(10);
		del = new DelegateType(MyFunction2);
		del(25);
		Console.ReadLine();
	}

	public static void MyFunction(int i)
	{
		Console.WriteLine("I'm in MyFunction and i={0}",i);
	}

	public static void MyFunction2(int i)
	{
		Console.WriteLine("I'm in MyFunction2 and i={0}",i);
	}
}
```

Explications :

- On déclare tout d'abord un ***delegate*** via le mot clef homonyme. On lui précise les arguments que la fonction déléguée devra accepter (ici un ***int***).
- On crée ensuite un objet de ce type en lui précisant quelle est la fonction appelée. On peut alors appeler la fonction désignée par l'intermédiaire de l'objet délégué.

11.3. Evénements

Les événements permettent de capturer une action du programme. Ainsi, lorsque l'on clique sur un bouton, l'événement ***Click*** du bouton est levé. On peut capturer les événements émis par les classes du Framework mais aussi créer ses propres événements.

Observons ce code :

```
// Events1.cs
using System;
```

```
public class Test
{
	public event EventHandler Changed;

	private string _msg;
	public string Msg
	{
		get
		{
			return _msg;
		}
		set
		{
			_msg=value;
			if (Changed != null) Changed(this,EventArgs.Empty);
		}
	}
}
public class TestHooking
{
	Test _test;
	public TestHooking()
	{
		_test = new Test();
		StartHooking();
	}
	public void Test_OnChanged(object sender,EventArgs e)
	{
		Console.WriteLine("I hook the Changed event from _test");
	}
	public void ChangeString(string Msg)
	{
		Console.WriteLine("Changing _test.Msg to {0}",Msg);
		_test.Msg=Msg;
	}
	public void StartHooking()
	{
		Console.WriteLine("Starting Hooking");
		_test.Changed += new EventHandler(Test_OnChanged);
	}
	public void StopHooking()
	{
		Console.WriteLine("Stoping Hooking");
		_test.Changed -= new EventHandler(Test_OnChanged);
	}
}
public class Events1
{
	public static void Main()
	{
		TestHooking th=new TestHooking();

		th.ChangeString("hello");
		th.StopHooking();
		th.ChangeString("Titi");
		th.StartHooking();
		th.ChangeString("Koala");
		Console.ReadLine();
	}
}
```

Résultat :

```
Starting Hooking
Changing _test.Msg to hello
I hook the Changed event from _test
Stoping Hooking
Changing _test.Msg to Titi
Starting Hooking
Changing _test.Msg to Koala
I hook the Changed event from _test
```

Explications :

- Un événement se déclare en plusieurs étapes. La première est de déclarer l'événement dans la classe en l'associant à un gestionnaire d'événement (***EventHandler***). Ce gestionnaire est de type ***delegate*** auquel nous pourrons associer une méthode qui sera donc le traitement de notre événement.
- Ainsi dans notre exemple, dans la classe ***TestHooking***, l'événement ***test.Changed*** est capturé via la ligne : test.Changed += new EventHandler(Test_OnChanged);
- En réalité, comme ***EventHandler*** est un type par délégation, la classe ***Test*** délègue une méthode. C'est pourquoi, pour lever l'événement ***Changed*** de la classe ***Test***, on teste d'abord si ***Changed*** est ***null*** (donc la délégation n'est pas effectuée) avec d'appeler la fonction déléguée.

Saisissez, compilez et exécutez le code suivant :

```
// Events2.cs

using System;

public delegate void FinishedCarEventHandler(
                  object sender, FinishedCarEventArgs e);

public class FinishedCarEventArgs : EventArgs
{
      private string _model;
      public string Model
      {
            get{ return _model; }
      }
      public FinishedCarEventArgs(string Model)
      {
            _model=Model;
      }
}

public class CarFactory
{
      public event FinishedCarEventHandler FinishedCar;

      public void BuildCar(string Model)
      {
            Console.WriteLine("I built a {0}",Model);
            if (FinishedCar != null)
            {
                  FinishedCar(this,new FinishedCarEventArgs(Model));
            }
      }
```

```
}

public class Events2
{
      public static void Main()
      {
            CarFactory cf=new CarFactory();

            cf.FinishedCar += new FinishedCarEventHandler(OnFinishedCar);
            cf.BuildCar("R5");
            cf.BuildCar("F350");
            cf.BuildCar("Laguna");

            Console.ReadLine();
      }

      public static void OnFinishedCar(
                  object sender,FinishedCarEventArgs e)
      {
            Console.WriteLine(
                 "The factory has finished building a {0}",e.Model);
      }
}
```

Sortie écran :

```
I built a R5
The factory has finished building a R5
I built a F350
The factory has finished building a F350
I built a Laguna
The factory has finished building a Laguna
```

Explications :

- Le principe est le même que l'exemple précédent, mais nous avons développé notre propre type d'événement.
- Nous avons pour cela créé un type par délégation pour capturer notre événement personnalisé. Chaque événement possède deux arguments : ***sender*** (***object***) qui est la source de l'événement, et ***e*** (***EventArgs***) qui sont les arguments de l'événement.
- Nous avons alors justement hérité la classe ***EventArgs*** pour créer nos propres arguments d'événement en rajoutant ici une chaîne de caractères ***Model***.

12. Propriétés et indexeurs

12.1. Propriétés

Qu'est ce qu'une propriété ?

Les propriétés s'utilisent comme une variable. La différence se situe au niveau de l'implémentation. En effet, plutôt que d'attribuer un espace mémoire et de le remplir avec une valeur, on va soi-même définir son comportement en lecture et/ou en écriture. On peut ainsi, pour chaque propriété définir une méthode ***get*** et une méthode ***set*** pour respectivement récupérer une valeur ou l'affecter.

Pour implémenter une propriété en lecture seule, on n'implémente que la méthode ***get***, et de même pour implémenter une propriété en écriture seule, on n'implémente que la propriété ***get***.

Afin de définir la valeur d'une variable, on utilise le paramètre ***value*** qui doit être du même type que la propriété.

Observons le code suivant :

```
// PropertySimple.cs

using System;

public class TestProp
{
	private int _updateCount;
	private int _accessCount;
	private int _value;

	public int Value
	{
		get
		{
			_accessCount++;
			return _value;
		}
		set
		{
			_updateCount++;
			_value=value;
		}
	}
	public override string ToString()
	{
		return string.Format(
    "My value is {0}, i've been accessed {1} times and updated {2} times.",
			_value,
			_accessCount,
			_updateCount
			);
	}
	public TestProp(int Value)
	{
		_accessCount=0;
		_updateCount=0;
		this.Value=Value;
	}
}
public class PropertySimple
{
```

```
      static void Main(string[] args)
      {
            TestProp tp=new TestProp(3);

            Console.WriteLine(tp.Value);
            tp.Value=4;
            Console.WriteLine(tp.Value);
            tp.Value=6;
            Console.WriteLine(tp.Value);
            tp.Value=-12;
            Console.WriteLine(tp.Value);
            tp.Value=806;
            tp.Value=32;
            tp.Value=1;
            Console.WriteLine(tp.Value);


            Console.WriteLine(tp.ToString());

            Console.ReadLine();
      }
}
```

Sortie écran :

```
3
4
6
-12
1
My value is 1, i've been accessed 5 times and updated 7 times.
```

Explications :

- Faire ***Console.WriteLine(tp.Value)***revient à appeler la méthode ***get*** de la propriété ***Value***. Dans cette propriété on a simplement retourné une valeur (qui doit d'ailleurs être du même type que la propriété).
- A l'opposé, faire ***tp.Value=32*** reviens à appeler la méthode set de la propriété ***Value*** en lui donnant comme argument 32 (que l'on récupère alors via le mot clef ***value***).

Modifions la classe ***Personne*** vue lors à l'exemple sur les classes. Pour cela modifiez le code pour obtenir :

```
// PersonApp2.cs
using System;

// notre classe exemple
public class Personne
{
      // L'âge
      private int _age;
      public int Age
      {
            get
            {
                  return _age;
            }
            set
```

```
            {
                  _age=value;
                  EvaluateThinking();
            }
      }
      // Le nom
      private string _name;
      public string Name
      {
            get
            {
                  return _name;
            }
            set
            {
                  _name=value;
                  EvaluateThinking();
            }
      }

      // Membre privé
      private string _thinkings;

      // Constructeurs
      public Personne(int Age,string Name)
      {
            this.Age=Age;
            this.Name=Name;
      }
      public Personne():this(0,"Unknown")
      {
      }

      // méthode privée
      private void EvaluateThinking()
      {
            this._thinkings=string.Format(
                        "My name is {0} and I'm {1} old",
                        Name,Age);
      }

      // méthode publique
      public void SayMyThinkings()
      {
            Console.WriteLine(_thinkings);
      }
}

// la classe contenant le point d'entrée Main
public class PersonApp
{
      public static void Main()
      {
            // Création de quelques personnes :
            Personne Charles=new Personne(25,"Charles");
            Personne Edward=new Personne(60,"Edward");
```

```
            Personne Inconnue=new Personne();

            // Faisons parler nos chères personnes.
            Charles.SayMyThinkings();
            Edward.SayMyThinkings();
            Inconnue.SayMyThinkings();

            // on fait plus appel ici aux membres publics
            // mais aux propriétés publiques
            Charles.Age++;
            Inconnue.Name="Robert";
            Inconnue.Age=10;

            // On leur redemande de dire leur pensée :
            Charles.SayMyThinkings();
            Edward.SayMyThinkings();
            Inconnue.SayMyThinkings();

            Console.ReadLine();
      }
}
```

Le résultat est identique à la version précédente, pourtant il y a une différence.

Explications :

- Comme je l'expliquais plus haut, on maîtrise l'affection ou la récupération des propriétés. Dans cette exemple, le but est de gérer deux membres privés de la classe Personne ***_age*** et ***_name***. La différence se situe lors de l'appel à la fonction ***EvaluateThinkings***. Elle ne se fait plus manuellement dès que l'on a besoin de ***_thinkings***, mais automatiquement dès que l'on met à jour le nom ou l'age.
- Comment se déroule le programme ? La ligne ***Inconnue.Name="Robert"*** fait appel à la fonction set de la propriété ***Name***. Cette dernière a alors deux instructions, d'une part, l'appel à la fonction ***EvaluateThinkings*** qui est nécessaire pour la mise à jour de ***_thinkings***, et d'autre part l'affectation de ***_name***. Remarquez le mot clef ***value*** qui indique la valeur appelée par le programmeur. Je vous conseille d'exécuter ce programme pas à pas pour bien visualiser l'ordre d'exécution.

12.2. Indexeurs

Les indexeurs sont comparables aux propriétés. Cependant un indexeur permet d'accéder, non pas à une seule variable comme à l'aide des propriétés mais à une collection de variables. Ainsi, la collection définie par la classe pourra être consultée ou modifiée à l'aide de l'opérateur ***[]***.

```
class Application
{
      static void Main(string[] args)
      {
            Eleve martin            = new Eleve("Martin",14);
            Eleve marc        = new Eleve("Marc",13);

            Classe      maclasse    = new Classe();
            maclasse.Add(martin);
            maclasse.Add(marc);

            Console.WriteLine(maclasse[0].Nom + "\t" + maclasse[0].Age);
            Console.WriteLine(maclasse[1].Nom + "\t" + maclasse[1].Age);
            Console.Read();
```

```
	}
}

public class Eleve
{
	private string	_nom;
	private int	_age;

	public Eleve(string nom,int age)
	{
		_nom  = nom;
		_age  = age;
	}

	public string Nom
	{
		get{return _nom;}
		set{_nom = value;}
	}
	public int Age
	{
		get{return _age;}
		set{_age = value;}
	}
}

public class Classe
{
	private ArrayList _eleves;

	public Classe ()
	{
		_eleves = new ArrayList();
	}

	public Eleve this[int index]
	{
		get{return (Eleve)_eleves[index];}
		set{_eleves[index] = value;}
	}

	public void Add(Eleve eleve)
	{
		_eleves.Add(eleve);
	}
}
```

Sortie écran :

```
Martin		14
Marc		13
```

Explications :

- Une classe représentant une classe d'élèves est définie. Cette classe contient donc une collection (***ArrayList***) d'élèves dont le type a été définie.
- L'indexeur permet d'accéder directement aux élèves de la classe.

13. Attributs

Un attribut permet de spécifier des paramètres supplémentaires à une méthode, une classe, une propriété. Ces paramètres peuvent servir à donner encore plus de détails dans les spécificités d'un objet, d'une classe, etc. que les simples ***public***, ***private***, etc. On peut par exemple, spécifier qu'une classe est visible depuis un WebService avec l'attribut ***[WebMethod]***, ou encore indiquer qu'une méthode s'exécute dans un même espace mémoire en cas d'utilisations de threads avec ***[StaThread]***.
Remarquez que les attributs se déclarent en précédent la méthode ou la classe rattaché par un [nomattribut1,nomattribut2,etc].
On peut également spécifier des paramètres à un attribut. En effet, chaque attribut est hérité de la classe ***System.Attribut*** et on peut donc créer des constructeurs spécialisés.

Exemple : utilisation d'un attribut existent

```
// attributes1.cs

using System;
using System.Diagnostics;

[Flags]
public enum StateFlags
{
      WithWheels= 0x0001,
      WithEngine= 0x0002,
      WithLights= 0x0004,
      WithBreaks= 0x0008
}

public class Attributes1
{
      public static void Main()
      {
            StateFlags State;

            State = StateFlags.WithBreaks;

            State |= StateFlags.WithEngine;

            if ((State & StateFlags.WithLights) != 0)
                  Console.WriteLine("Yes, there are lights installed");
            else
                  Console.WriteLine("No, there is no light installed");

            Console.WriteLine("CurrentFlags are {0}",State);

            Console.ReadLine();
      }
}
```

Sortie écran :

```
No, there is no light installed
CurrentFlags are WithEngine, WithBreaks
```

Explications :

- Ici nous avons via l'attribut ***[Flags]*** (dont l'applicabilité est sur les énumérations exclusivement) indiqué au compilateur qu'il doit traiter notre énumération comme un champ de bit, et non une valeur entière.
- Le fonctionnement de la fonction diffère sans cet attribut. Essayez de supprimer l'attribut, et vous verrez. Même si le programme fonctionne, il sera incapable d'afficher correctement ***State***. Si le flag est mis, il reconnaîtra la combinaison des deux valeurs.
- On voit donc comment avec les attributs on peut influencer le déroulement du programme.