

Introduction

Table des matières

Introduction	1
Documents de cours	2
Conditions préalables	3
Plan du cours	4
Programme MCP	7
Logistique	10



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Introduction

- Nom
- Affiliation à une société
- Titre / Fonction
- Responsabilité professionnelle
- Expérience en programmation
- Expérience en C, C++, Visual Basic ou Java
- Attentes vis-à-vis du cours

Documents de cours

- Fiche
- Manuel de travail du stagiaire
- CD-ROM du stagiaire
- Évaluation du cours

Votre kit de cours contient les documents détaillés ci-dessous :

- *Fiche*. Inscrivez votre nom des deux côtés de la fiche.
- *Manuel de travail du stagiaire*. Ce manuel expose les sujets traités dans le cadre du cours, ainsi que les exercices réalisés pendant les ateliers.
- *CD-ROM du stagiaire*. Ce CD-ROM contient la page Web destinée aux stagiaires. Cette page comporte des liens renvoyant à des ressources ayant un rapport avec ce cours, notamment des lectures complémentaires, les réponses aux questions de contrôle des acquis et des ateliers, les fichiers des ateliers, les présentations multimédias et les sites Web se rapportant au cours.

Remarque Pour ouvrir la page Web du stagiaire, insérez le CD-ROM du stagiaire dans le lecteur, puis double-cliquez à la racine sur **Default.htm** ou **Autorun.exe**.

- *Évaluation du cours*. Pour envoyer vos commentaires et impressions sur l'instructeur, le cours et le produit logiciel exploité, envoyez un courrier électronique à l'adresse de messagerie support@mscourseware.com ; veuillez à taper **Cours 2124C (2132A)** dans la zone Objet de votre message. Cette adresse correspondant à un service américain de Microsoft, pensez à rédiger votre message en anglais. Vos commentaires nous aideront à améliorer nos prochains cours. Pour formuler des commentaires supplémentaires ou obtenir des renseignements sur le programme MCP, vous pouvez envoyer un courrier électronique à l'adresse mcphelp@microsoft.com. Ce service est uniquement disponible en anglais.

Conditions préalables

- **Expérience de la programmation en C, C++, Visual Basic ou Java**
- **Connaissance de la stratégie Microsoft .NET**
- **Connaissance de Microsoft .NET Framework**

Pour tirer pleinement parti de ce cours, vous devez disposer des connaissances suivantes :

- avoir de l'expérience en matière de programmation en C, C++, Microsoft® Visual Basic®, Java ou autre langage de programmation ;
- être familiarisés avec la stratégie Microsoft .NET décrite sur le site Web de Microsoft .NET (<http://www.microsoft.com/france/net>) ;
- être familiarisés avec le .NET Framework décrit sur le site Web de Microsoft MSDN® : <http://www.microsoft.com/france/msdn/technologies/technos/framework/>.

Plan du cours

- **Module 1 : Vue d'ensemble de la plate-forme Microsoft .NET**
- **Module 2 : Vue d'ensemble de C#**
- **Module 3 : Utilisation des variables de type valeur**
- **Module 4 : Instructions et exceptions**
- **Module 5 : Méthodes et paramètres**

Le module 1, « Vue d'ensemble de la plate-forme Microsoft .NET », décrit la structure et les fonctionnalités qui sous-tendent la plate-forme .NET, y compris les composants .NET. Ce module a pour objectif de vous faire comprendre la plate-forme .NET pour laquelle vous allez développer du code en C#. À la fin de ce module, vous serez à même de décrire les composants de la plate-forme .NET.

Le module 2, « Vue d'ensemble de C# », décrit la structure de base d'une application développée en C#. Ce module présente un exemple fonctionnel simple dont l'analyse vous permet d'apprendre à utiliser la classe **Console** pour effectuer des opérations d'entrée et de sortie de base et enfin, de découvrir les méthodes conseillées pour gérer les erreurs et documenter votre code. À la fin de ce module, vous serez à même de compiler, d'exécuter et de corriger une application développée en C#.

Le module 3, « Utilisation des variables de type valeur », décrit comment utiliser des variables de type valeur en C#. Ce module explique comment spécifier le type des données contenues dans les variables, nommer les variables d'après les conventions d'affectation de noms standard, assigner des valeurs aux variables et enfin, convertir des variables d'un type de données vers un autre. À la fin de ce module, vous serez à même d'utiliser des variables de type valeur en C#.

Le module 4, « Instructions et exceptions », explique comment utiliser certaines instructions courantes en C#. Ce module décrit également comment implémenter la gestion des exceptions en C#. À la fin de ce module, vous serez à même de lever et d'intercepter des erreurs.

Le module 5, « Méthodes et paramètres », décrit comment créer des méthodes statiques qui prennent des paramètres et retournent des valeurs, passer des paramètres à des méthodes de différentes façons et enfin, déclarer et utiliser des méthodes surchargées. À la fin de ce module, vous serez à même d'utiliser des méthodes et des paramètres.

Plan du cours (*suite*)

- **Module 6 : Tableaux**
- **Module 7 : Notions fondamentales de la programmation orientée objet**
- **Module 8 : Utilisation des variables de type référence**
- **Module 9 : Création et destruction d'objets**
- **Module 10 : Héritage dans C#**

Le module 6, « Tableaux », explique comment grouper des données dans des tableaux. À la fin de ce module, vous serez à même de créer, d'initialiser et d'utiliser des tableaux.

Le module 7, « Notions fondamentales de la programmation orientée objet », présente la terminologie et les concepts requis pour créer et utiliser des classes en C#. Ce module décrit également l'abstraction, l'encapsulation, l'héritage et le polymorphisme. À la fin de ce module, vous serez à même d'expliquer certains des concepts les plus courants de la programmation orientée objet.

Le module 8, « Utilisation des variables de type référence », décrit comment utiliser des variables de type référence en C#. Ce module présente plusieurs types référence, tels que string, qui sont intégrés dans le langage C# et dans le Common Language Runtime. À la fin de module, vous serez à même d'utiliser des variables de type référence en C#.

Le module 9, « Création et destructions d'objets », explique ce qui se produit dans le runtime lorsqu'un objet est créé et comment utiliser des constructeurs pour initialiser des objets. Ce module explique également ce qui se produit lorsqu'un objet est détruit et comment le garbage collector récupère la mémoire. À la fin de ce module, vous serez à même de créer et de détruire des objets en C#.

Le module 10, « Héritage dans C# », explique comment dériver une classe d'une autre classe. Il explique également comment implémenter des méthodes dans une classe dérivée en les définissant en tant que méthodes virtuelles dans la classe de base et en les substituant ou en les masquant dans la classe dérivée, le cas échéant. Enfin, ce module explique comment sceller une classe pour qu'il soit impossible d'en dériver une autre et comment implémenter des interfaces et des classes abstraites. À la fin de ce module, vous serez à même d'utiliser l'héritage en C# pour dériver des classes et de définir des méthodes virtuelles.

Plan du cours (*suite*)

- **Module 11 : Agrégation, espaces de noms et portée avancée**
- **Module 12 : Opérateurs, délégués et événements**
- **Module 13 : Propriétés et indexeurs**
- **Module 14 : Attributs**
- **Annexe A : Ressources pour une étude approfondie**

Le module 11, « Agrégation, espaces de noms et portée avancée », décrit comment regrouper des classes en niveaux plus importants et plus élevés de classes et utiliser les espaces de noms pour y regrouper des classes et créer des structures de programmes logiques supérieures aux classes individuelles. Ce module explique également comment utiliser les assemblés pour regrouper des fichiers sources collaborant entre eux dans une unité réutilisable, déployable et pouvant être associée à un numéro de version. À la fin de ce module, vous serez à même de rendre du code accessible au niveau composant ou assembly.

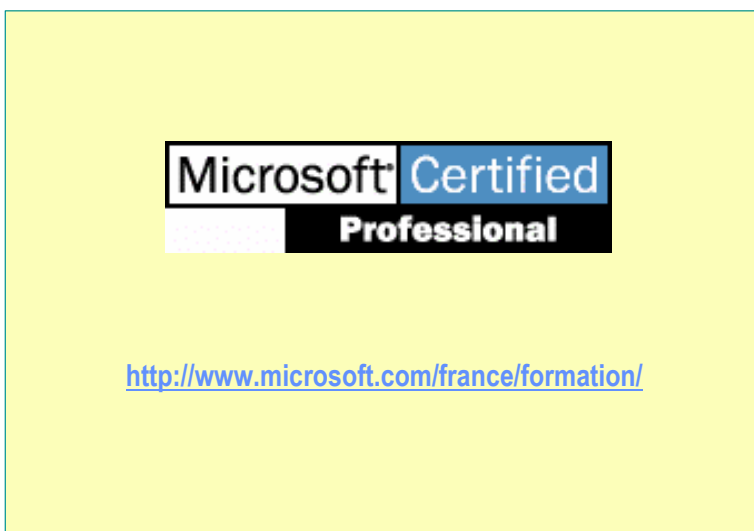
Le module 12, « Opérateurs, délégués et événements », explique comment définir des opérateurs et utiliser des délégués pour dissocier un appel de méthode de son implémentation. Il explique également comment ajouter des spécifications d'événements à une classe. À la fin de ce module, vous serez à même d'implémenter des opérateurs, des délégués et des événements.

Le module 13, « Propriétés et indexeurs », explique comment créer des propriétés pour encapsuler des données dans une classe et définir des indexeurs pour accéder aux classes en utilisant une notation de type tableau. À la fin de ce module, vous serez à même d'utiliser des propriétés pour permettre un accès de type champ et des indexeurs pour permettre des accès de type tableau.

Le module 14, « Attributs », décrit le rôle des attributs et la fonction qu'ils assurent dans les applications C#. Ce module présente la syntaxe des attributs et explique comment utiliser certains attributs prédéfinis dans l'environnement .NET. À la fin de ce module, vous serez à même de créer des attributs personnalisés définis par l'utilisateur et d'utiliser ces attributs personnalisés pour obtenir des informations d'attribut au moment de l'exécution.

L'annexe A, « Ressources pour une étude approfondie », est un document qui sert de référence une fois le cours terminé pour approfondir les connaissances acquises et vous aider à rechercher les toutes dernières informations sur le langage de programmation C# et le .NET Framework.

Programme MCP



Le programme MCP (*Microsoft Certified Professional*) est un programme de certification renommé qui valide votre expérience et vos connaissances pour assurer votre compétitivité sur le marché de l'emploi. Le tableau suivant décrit chacune des certifications.

Certification	Description
MCSA sur Microsoft Windows 2000	La certification MCSA (<i>Microsoft Certified Systems Administrator</i>) est destinée aux professionnels qui implémentent, gèrent et dépannent des environnements réseau et système existants fondés sur les plates-formes Microsoft Windows 2000, dont la famille Windows .NET Server. Leurs tâches d'implémentation comprennent l'installation et la configuration d'au moins une partie de ces systèmes. En matière de gestion, leurs responsabilités englobent l'administration et le support technique de ces systèmes.
MCSE sur Microsoft Windows 2000	La certification MCSE (<i>Microsoft Certified Systems Engineer</i>) est la principale certification des professionnels dont le rôle consiste à analyser les besoins des entreprises, pour ensuite concevoir et implémenter des infrastructures de solutions d'entreprise fondées sur la plate-forme Microsoft Windows 2000 et les logiciels serveurs de Microsoft, dont la famille Windows .NET Server. Leurs tâches d'implémentation comprennent l'installation, la configuration et le dépannage de systèmes réseau.
MCSD	La certification MCSD (<i>Microsoft Certified Solution Developer</i>) est la principale certification pour les professionnels qui conçoivent et développent des solutions d'entreprise de pointe à l'aide d'outils de développement, de technologies, de plates-formes Microsoft et de l'architecture Microsoft Windows DNA. Parmi les types d'applications que les titulaires d'une certification MCSD sont susceptibles de développer, citons les applications de bureau et les applications multi-utilisateurs, Web, multiniveaux et transactionnelles. Les tâches qu'ils assument vont de l'analyse des besoins de l'entreprise à la maintenance des solutions mises en œuvre.

(suite)

Certification	Description
MCDBA sur Microsoft SQL Server™ 2000	La certification MCDBA (<i>Microsoft Certified Database Administrator</i>) est la principale certification à l'intention des professionnels chargés de l'implémentation et de l'administration de bases de données Microsoft SQL Server. Cette certification valide les compétences dans les domaines suivants : mise à jour d'anciens modèles de bases de données physiques, développement de modèles de données logiques, création de bases de données physiques, création de services de données au moyen de Transact-SQL, gestion et maintenance des bases de données, configuration et gestion de la sécurité, surveillance et optimisation des bases de données, et installation et configuration de SQL Server.
MCP	La certification MCP (<i>Microsoft Certified Professional</i>) s'adresse aux individus qui possèdent les compétences nécessaires pour implémenter un produit ou une technologie Microsoft au sein d'une solution d'entreprise dans une organisation. Il est nécessaire de bénéficier d'une expérience pratique du produit pour obtenir cette certification.
MCT	La certification MCT (<i>Microsoft Certified Trainer</i>) valide les compétences pédagogiques et techniques des personnes qui dispensent des cours MOC (<i>Microsoft Official Curriculum</i>) dans les centres de formation technique agréés Microsoft CTEC (<i>Certified Technical Education Center</i>).

Conditions requises pour l'obtention des certifications

Les critères retenus varient en fonction des certifications. Ils dépendent des produits et des responsabilités professionnelles auxquels répond chacune de ces certifications. Pour devenir MCP, vous devez être reçu à des examens de certification rigoureux qui permettent d'évaluer de manière valide et fiable vos compétences et connaissances techniques.

Pour plus d'informations Consultez le site Web Formations et Certifications de Microsoft France à l'adresse <http://www.microsoft.com/france/formation/>.

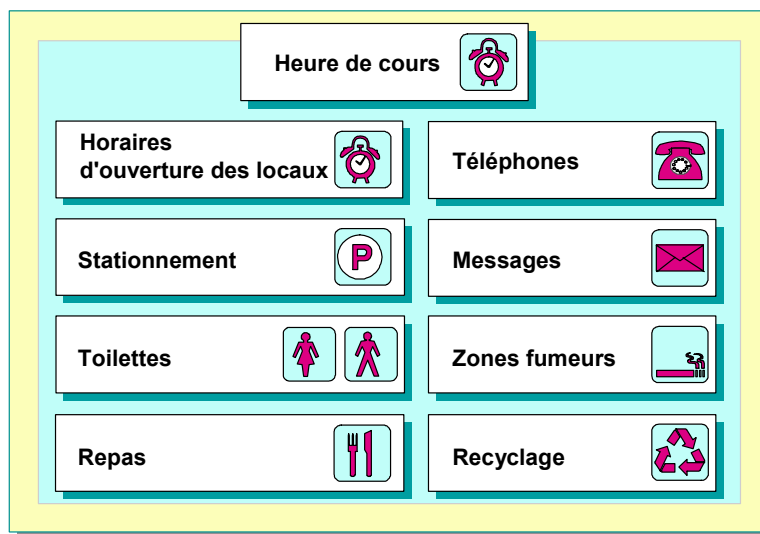
Vous pouvez aussi envoyer un courrier électronique à l'adresse mcphelp@microsoft.com pour toute question concernant les certifications.

Acquisition des compétences validées par un examen MCP

Les cours MOC (*Microsoft Official Curriculum*) et MSDN Training Curriculum peuvent vous aider à développer les compétences nécessaires pour assumer vos fonctions. Ils renforcent également l'expérience que vous avez acquise lors de l'utilisation des produits et technologies Microsoft. Cependant, il n'existe pas de correspondance directe entre les cours de formation MOC et MSDN et les examens MCP. Le seul suivi des cours ne garantit pas nécessairement la réussite aux examens MCP : de l'expérience et des connaissances pratiques sont également nécessaires.

Pour préparer les examens MCP, vous pouvez utiliser les guides de préparation disponibles pour chaque examen. Chaque Guide de préparation contient des informations spécifiques à un examen, telles que la liste des sujets sur lesquels vous serez interrogé. Ces guides sont disponibles sur le site Web Formations et Certifications de Microsoft France à l'adresse <http://www.microsoft.com/france/formation/>.

Logistique



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

Table des matières

Vue d'ensemble	1
Présentation de la plate-forme .NET	2
Vue d'ensemble du .NET Framework	5
Avantages du .NET Framework	7
Composants du .NET Framework	8
Langages du .NET Framework	14
Contrôle des acquis	16



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Présentation de la plate-forme .NET
- Vue d'ensemble du .NET Framework
- Avantages du .NET Framework
- Composants du .NET Framework
- Langages du .NET Framework

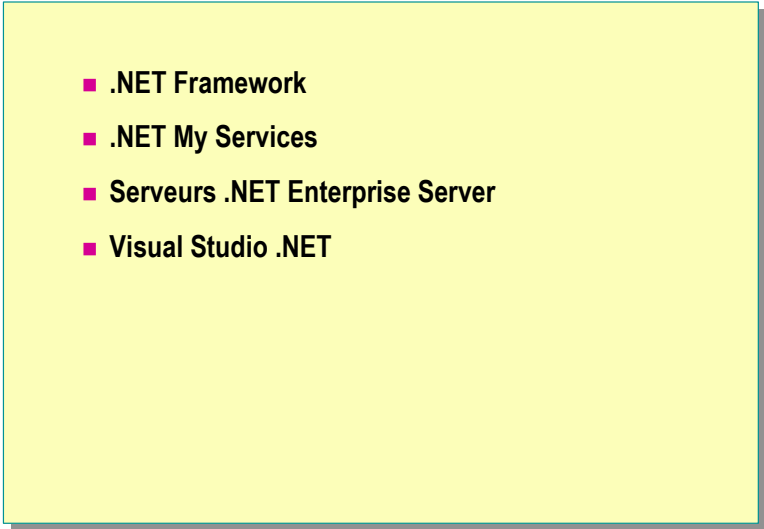
La plate-forme Microsoft® .NET fournit l'ensemble des outils et technologies nécessaires à la création d'applications Web distribuées. Elle expose un modèle de programmation cohérent, indépendant du langage, à tous les niveaux d'une application, tout en garantissant une parfaite interopérabilité avec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

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- décrire la plate-forme .NET ;
- citer les principaux éléments de la plate-forme .NET ;
- décrire le .NET Framework et ses composants ;
- expliquer la prise en charge des langages de programmation par le .NET Framework .

Présentation de la plate-forme .NET

- 
- .NET Framework
 - .NET My Services
 - Serveurs .NET Enterprise Server
 - Visual Studio .NET

La plate-forme .NET offre plusieurs technologies de base, comme l'illustre cette diapositive. Ces technologies sont décrites dans les rubriques suivantes.

.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.

.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

Serveurs .NET Enterprise 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.

Serveur	Description
Microsoft SQL Server™	Offre une fonctionnalité XML riche ; une prise en charge des standards W3C (<i>Worldwide Web Consortium</i>) ; la capacité à manipuler des données XML au moyen de Transact SQL (T-SQL) ; une analyse Web flexible et puissante ; un accès sécurisé aux données sur le Web au moyen de HTTP.
Microsoft BizTalk™ Server	Assure l'intégration d'applications d'entreprise (EAI, <i>Enterprise Application Integration</i>) et l'intégration interentreprises ; fournit la technologie avancée BizTalk Orchestration pour l'élaboration de processus métiers dynamiques s'appliquant aux applications, plates-formes et organisations via Internet.
Microsoft Host Integration Server	Constitue la meilleure manière d'adopter les technologies Internet, intranet et client-serveur tout en préservant les investissements effectués dans des systèmes antérieurs existants.
Microsoft Exchange Enterprise Server	Exploite la puissante technologie de messagerie et de collaboration d'Exchange et introduit plusieurs nouvelles fonctionnalités importantes, améliorant ainsi la fiabilité, l'évolutivité et les performances de son architecture de base. D'autres fonctionnalités améliorent l'intégration d'Exchange avec Microsoft Windows®, Microsoft Office et Internet.
Microsoft Application Center	Constitue un outil de déploiement et de gestion pour les applications Web à haute disponibilité.
Microsoft Internet Security and Acceleration Server	Offre une connectivité Internet sécurisée, rapide et facile à gérer. Internet Security and Acceleration Server intègre un pare-feu d'entreprise extensible et multicouche et un cache Web évolutif, hautement performant. Il s'appuie sur l'annuaire et la sécurité Windows pour assurer la sécurité fondée sur les stratégies, l'accélération et la gestion de la mise en réseau.
Microsoft Commerce Server	Fournit un cadre d'applications, des mécanismes de rétroaction sophistiqués et des capacités d'analyse.

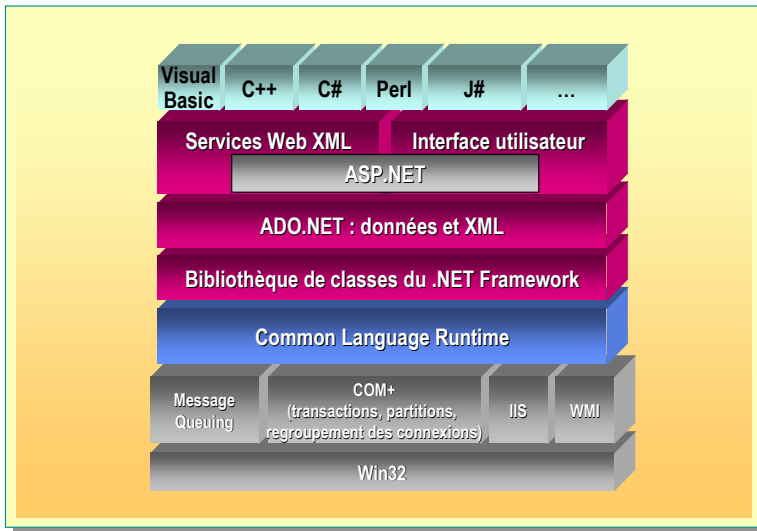
(suite)

Serveur	Description
Microsoft SharePoint™ Portal Server	Permet de créer des portails Web d'entreprise avec des fonctions de gestion de documents, recherche de contenu et collaboration d'équipe.
Microsoft Mobile Information Server	Intégré aux serveurs Microsoft .NET Enterprise Server et à Microsoft Windows pour fournir des communications sécurisées et permettre l'échange des données avec des périphériques mobiles. Il offre une fiabilité, une flexibilité et des performances optimales grâce aux fonctions de clustering, de réplication, d'équilibrage de la charge et de diffusion du contenu.
Microsoft Content Management Server	Offre un jeu complet de fonctions pour la distribution et la diffusion du contenu, le développement de sites et la gestion de sites d'entreprise afin de permettre aux entreprises de créer, déployer et gérer efficacement des sites Web Internet, intranet et extranet.

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.

Vue d'ensemble du .NET Framework



.NET Framework

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

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.

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.

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.

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.

Bibliothèque de classes du .NET Framework

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.

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.

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.

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.

Interfaces utilisateur

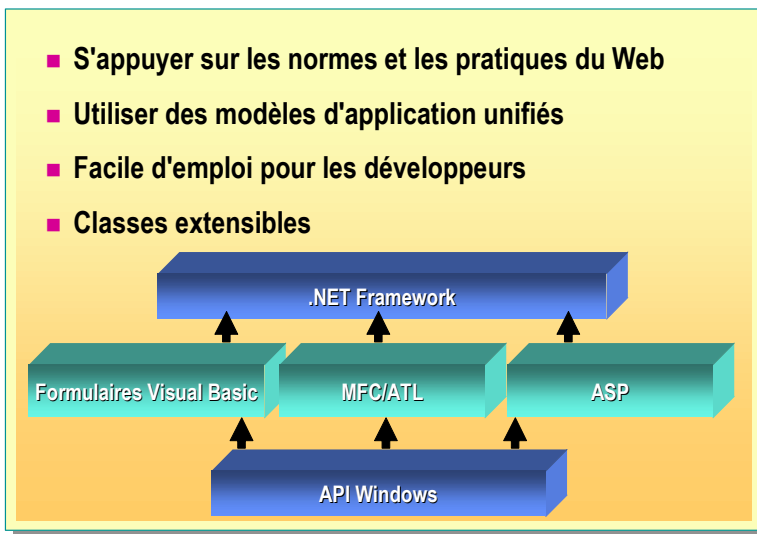
Le .NET Framework prend en charge trois types d'interfaces utilisateur :

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

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.

Avantages du .NET Framework



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

- **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.

- **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.

- **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.

- **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.

◆ Composants du .NET Framework

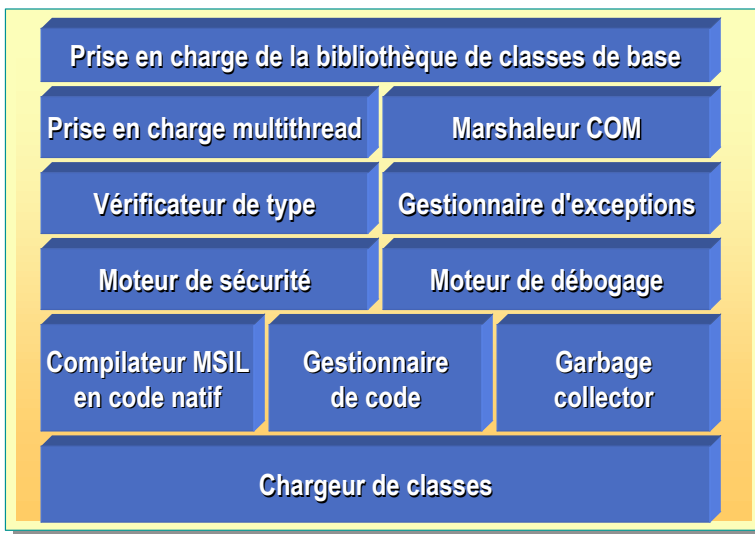
- Common Language Runtime
- Bibliothèque de classes du .NET Framework
- ADO.NET : données et XML
- Formulaires Web et les services Web XML
- Interface utilisateur pour Windows

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.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- décrire le Common Language Runtime ;
- décrire la bibliothèque de classes de base (BCL, *Base Class Library*) ;
- décrire les formulaires Web et les services Web XML ;
- utiliser l'interface utilisateur.

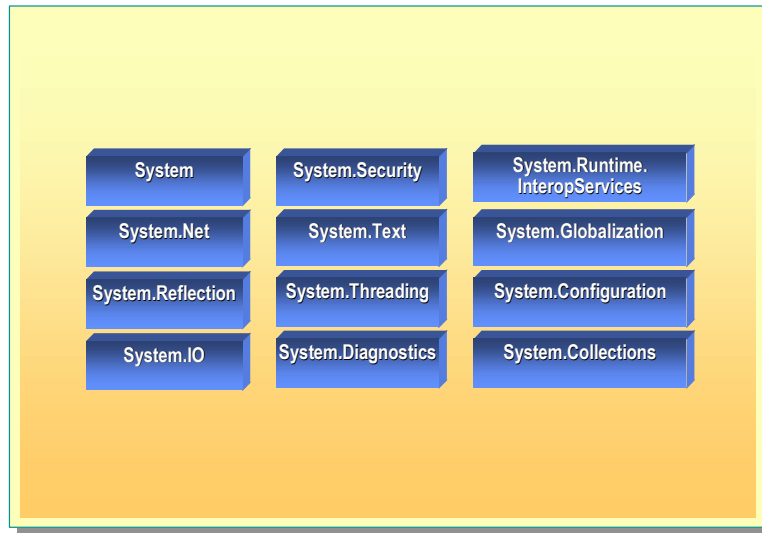
Common Language Runtime



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.

Composant	Description
Chargeur de classes	Gère les métadonnées, en plus du chargement des classes et de leur disposition.
Compilateur MSIL (<i>Microsoft Intermediate Language</i>) 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 (<i>Windows Structured Exception Handling</i>). 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.

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.

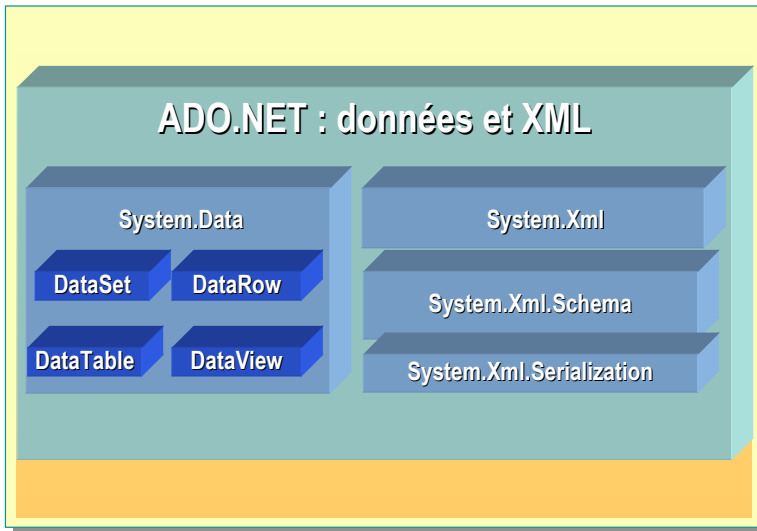
Espace de noms System

L'espace de noms **System** contient des classes fondamentales et de base qui définissent les types de données valeur et référence, les événements et gestionnaires d'événements, les interfaces, les attributs et les exceptions de traitement 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*).

Pour plus d'informations sur les espaces de noms, consultez la documentation du Microsoft .NET Framework SDK.

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**.

Espace de noms System.Data

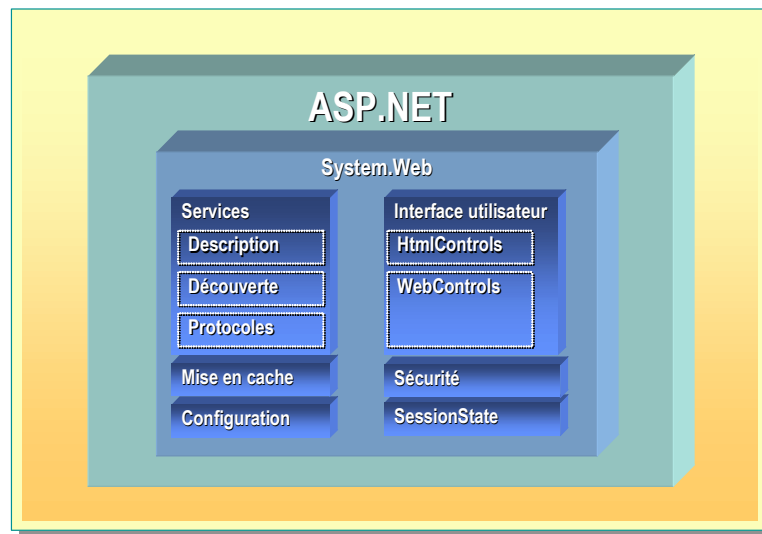
L'espace de noms **System.Data** est constitué de classes qui forment le modèle objet d'ADO.NET. À haut niveau, le modèle objet ADO.NET se divise en deux couches : la couche connectée et la couche déconnectée.

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.

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.

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.

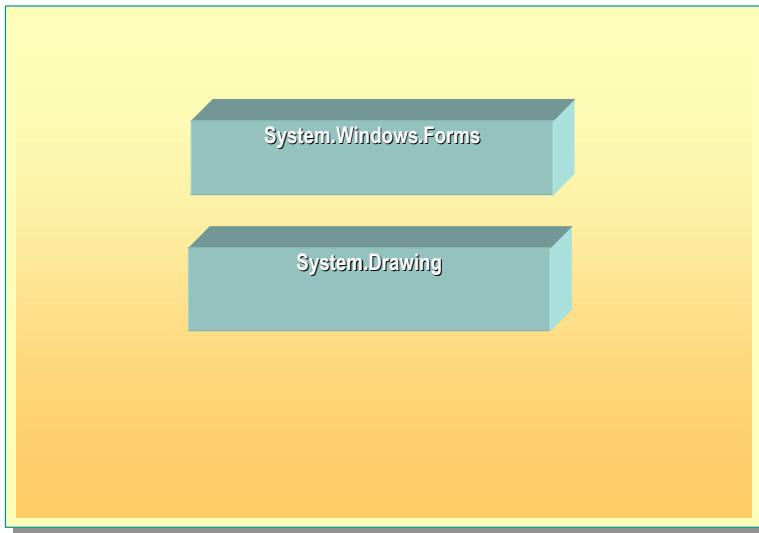
System.Web.Services

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

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.

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** et **System.Drawing.Text**.

Langages du .NET Framework

- **C# Conçu pour .NET**
Nouveau langage orienté composant
- **Extensions managées pour C++**
Améliorées pour plus de puissance et de contrôle
- **Visual Basic .NET**
Nouvelle version de Visual Basic comportant d'importantes innovations au niveau du langage
- **JScript .NET**
Nouvelle version de JScript améliorant les performances et la productivité
- **J# .NET**
Prise en charge du langage Java par .NET permettant un nouveau développement et la migration Java
- **Langages tiers**

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 (<i>Interface Definition Language</i>) 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.

(suite)

Langage	Description
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 (<i>Java User Migration Path to Microsoft .NET</i> , « <i>JUMP to .NET</i> »).
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.

Contrôle des acquis

- Présentation de la plate-forme .NET
- Vue d'ensemble du .NET Framework
- Avantages du .NET Framework
- Composants du .NET Framework
- Langages du .NET Framework

-
1. En quoi consiste la plate-forme .NET ?
 2. Quelles sont les technologies de base de la plate-forme .NET ?
 3. Citez les composants du .NET Framework.
 4. À quoi sert le Common Language Runtime ?

-
5. À quoi sert la spécification CLS ?
 6. Qu'est-ce qu'un service Web XML ?
 7. Qu'est-ce qu'un environnement managé ?

Module 2 : Vue d'ensemble de C#

Table des matières

Vue d'ensemble	1
Structure d'un programme en C#	2
Opérations élémentaires d'entrée/sortie	9
Méthodes conseillées	15
Compilation, exécution et débogage	22
Atelier 2.1 : Création d'un programme simple en C#	36
Contrôle des acquis	44



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Structure d'un programme en C#
- Opérations élémentaires d'entrée/sortie
- Méthodes conseillées
- Compilation, exécution et débogage

Dans ce module, vous allez apprendre la structure de base d'un programme C# à travers l'analyse d'un exemple de travail simple. Vous apprendrez à utiliser la classe **Console** pour effectuer des opérations élémentaires d'entrée/sortie. Vous découvrirez également quelques-unes des méthodes conseillées pour la gestion des erreurs et la documentation du code. Enfin, vous compilerez, exécuterez et déboguerez un programme C#.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- expliquer la structure d'un programme simple en C# ;
- utiliser la classe **Console** de l'espace de noms **System** pour effectuer des opérations d'entrée/sortie élémentaires ;
- gérer les exceptions dans un programme C# ;
- créer une documentation en XML (Extensible Markup Language) pour un programme C# ;
- compiler et exécuter un programme en C# ;
- utiliser le débogueur pour suivre l'exécution d'un programme.

◆ Structure d'un programme en C#

- Hello, World
- La classe
- La méthode Main
- La directive using et l'espace de noms System
- Démonstration : Utilisation de Visual Studio pour créer un 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#.

Hello, World

```
using System;

class Hello
{
    public static void Main()
    {
        Console.WriteLine("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#.

La classe

- Une application C# est une collection de classes, de structures et de types
- Une classe est un ensemble de données et de méthodes
- Syntaxe

```
class nom
{
    ...
}
```

- Une application C# peut être composée de nombreux fichiers
- Une classe ne peut pas s'étendre sur plusieurs fichiers

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.

Note pour les développeurs Java Le nom du fichier d'application ne doit pas nécessairement être le même que celui de la classe.

Note pour les développeurs C++ C# ne distingue pas la définition et l'implémentation d'une classe de la même manière que C++. Il n'existe pas de concept de fichier de définition (.hpp). Tout le code de la classe est écrit dans un même fichier.

La méthode Main

- **En écrivant Main, vous devez :**
 - utiliser un « M » majuscule, comme dans « Main » ;
 - désigner une méthode **Main** comme point d'entrée du programme ;
 - déclarer **Main** en tant que **public static void Main**.
- **Plusieurs classes peuvent avoir une méthode Main**
- **Dès que Main a fini ou qu'elle renvoie une instruction return, l'application se termine**

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.

Important 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.

Conseil 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**.

La directive using et l'espace de noms System

- Le .NET Framework offre de nombreuses classes d'utilitaires
 - Elles sont organisées en espaces de noms
- System est l'espace de noms le plus courant
- Faites référence aux classes à l'aide de leur espace de noms

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

■ La directive using

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

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.

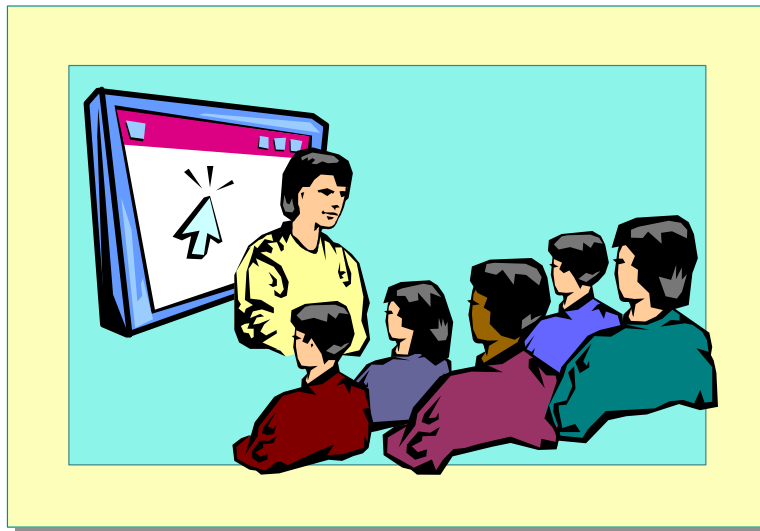
Avec la directive **using**, vous pouvez réécrire le code précédent de la manière suivante :

```
using System;  
...  
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**.

Remarque Les classes de l'espace de noms **System** et les autres fonctions principales utilisées au moment de l'exécution se trouvent dans un assembly appelé `mscorlib.dll`. Cet assembly est utilisé par défaut. Vous pouvez faire référence aux classes dans d'autres assemblies, mais vous devez alors spécifier les emplacements et les noms de ces assemblies lors de la compilation de l'application.

Démonstration : Utilisation de Visual Studio pour créer un programme C#



Dans cette démonstration, vous allez apprendre à utiliser Visual Studio pour créer et modifier un programme C#.

◆ Opérations élémentaires d'entrée/sortie

- La classe **Console**
- Méthodes **Write** et **WriteLine**
- Méthodes **Read** et **ReadLine**

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**.

La classe Console

- Permet d'accéder à l'entrée standard, à la sortie standard et aux flux d'erreur standard
- S'applique uniquement aux applications console
 - Entrée standard clavier
 - Sortie standard écran
 - Erreur standard écran
- Tous les flux peuvent être redirigés

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.

Remarque Ces flux et la classe **Console** s'appliquent uniquement aux applications console. Ces applications s'exécutent dans une fenêtre de commande.

Vous pouvez rediriger ces trois types de flux (entrées standard, sorties standard, erreurs standard) vers un fichier ou un périphérique. Vous pouvez effectuer cette opération par programme, ou l'utilisateur peut s'en charger lors de l'exécution de l'application.

Méthodes Write et WriteLine

- **Console.Write et Console.WriteLine permettent d'afficher des informations sur l'écran de la console**
 - **WriteLine** ajoute une nouvelle paire ligne/retour chariot
- **Ces deux méthodes sont surchargées**
- **Une chaîne de format et des paramètres peuvent être utilisés**
 - Formatage de texte
 - Formatage numérique

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");
```

Formatage de texte

Vous pouvez utiliser des formes plus puissantes des méthodes **Write** et **WriteLine**, qui utilisent une chaîne de format et des paramètres supplémentaires. La chaîne de format, qui spécifie le mode de sortie des données, peut contenir des marqueurs, qui sont remplacés dans l'ordre par les paramètres ci-dessous. Par exemple, vous pouvez utiliser le code suivant pour afficher le message « la somme de 100 et 130 est 230 » :

```
Console.WriteLine("La somme de {0} et {1} est {2}", 100, 130, 100+130);
```

Important Le premier paramètre qui suit la chaîne de format est appelé paramètre zéro : {0}.

Vous pouvez utiliser le paramètre de chaîne de format pour spécifier la largeur des champs et indiquer si les valeurs doivent être justifiées à gauche ou à droite dans ces champs, comme l'illustre le code suivant :

```
Console.WriteLine("\Justifié à gauche dans un champ de  
largeur 10 : {0, -10}\\"", 99);  
Console.WriteLine("\Justifié à droite dans un champ de  
largeur 10 : {0,10}\\"", 99);
```

Le code suivant s'affiche sur la console :

```
« Justifié à gauche dans un champ de largeur 10 : 99      »  
« Justifié à droite dans un champ de largeur 10 :      99 »
```

Remarque Vous pouvez utiliser la barre oblique inverse (\) dans une chaîne de format pour annuler la signification particulière du caractère suivant. Par exemple, « \{ » affiche un « { » littéral, et « \\ » affiche un « \ » littéral. Vous pouvez utiliser le *à commercial* (@) pour représenter la syntaxe entière d'une chaîne. Par exemple, @"\\serveur\partage" est traité comme "\\serveur\partage."

Formatage numérique

Vous pouvez également utiliser une chaîne de format pour spécifier le mode de formatage des données numériques. La syntaxe complète de la chaîne de format est {N,M:FormatString}, où N correspond au nombre de paramètres, M à la largeur et la justification du champ et FormatString au mode d'affichage des données numériques. Le tableau ci-dessous répertorie les éléments de la méthode **FormatString**. Dans tous ces formats, le nombre de décimales à afficher, ou à arrondir, peut être spécifié de manière facultative.

Élément	Signification
C	Afficher le nombre en tant qu'unité monétaire, en utilisant le symbole et les conventions monétaires locales.
D	Afficher le nombre en tant que nombre entier décimal.
E	Afficher le nombre en utilisant une notation exponentielle (scientifique).
F	Afficher le nombre en tant que valeur à virgule fixe.
G	Afficher le nombre en tant que nombre à virgule fixe ou nombre entier, en fonction du format le plus compact.
N	Afficher le nombre avec des séparateurs incorporés.
X	Afficher le nombre en utilisant une notation hexadécimale.

Le code suivant montre des exemples d'utilisation du formatage numérique :

```
Console.WriteLine("Formatage monétaire - {0:C} {1:C4}",  
    ↪88.8, -888.8);  
Console.WriteLine("Formatage de nombre entier - {0:D5}", 88);  
Console.WriteLine("Formatage exponentiel - {0:E}", 888.8);  
Console.WriteLine("Formatage à virgule fixe - {0:F3}",  
    ↪888.8888);  
Console.WriteLine("Formatage général - {0:G}", 888.8888);  
Console.WriteLine("Formatage de nombre - {0:N}", 8888888.8);  
Console.WriteLine("Formatage hexadécimal - {0:X4}", 88);
```

Lorsque le code précédent est exécuté, il affiche ce qui suit :

```
Formatage monétaire - 88,80 € (888,8000 €)  
Formatage de nombre entier - 00088  
Formatage exponentiel - 8,888000E+002  
Formatage à virgule fixe - 888,889  
Formatage général - 888,8888  
Formatage de nombre - 8 888 888,80  
Formatage hexadécimal - 0058
```

Remarque Pour plus d'informations sur le formatage, recherchez « spécificateurs de format » ou consultez ms-help://MS.VSCC/MS.MSDNVS.1036/cpguide/html/cpconformattingoverview.htm dans Microsoft Visual Studio .NET.

Méthodes Read et ReadLine

- **Console.Read et Console.ReadLine lisent l'entrée de l'utilisateur**
 - **Read** lit le caractère suivant
 - **ReadLine** lit la totalité de la ligne d'entrée

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

La méthode Read

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

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("{0}", input);
```


◆ Méthodes conseillées

- **Commentaire des applications**
- **Création d'une documentation XML**
- **Démonstration : Génération et affichage de la documentation XML**
- **Gestion des exceptions**

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.

Commentaire des applications

■ Les commentaires sont importants

- Une application bien commentée permet à un développeur de bien comprendre la structure de l'application

■ Commentaires sur une seule ligne

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

■ Commentaires sur plusieurs lignes

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

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.

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.

Création d'une documentation XML

```
/// <summary> La classe Hello affiche un message  
/// de bienvenue à l'écran  
/// </summary>  
class Hello  
{  
    /// <remarks> On utilise 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");  
    }  
}
```

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.

(suite)

Balise	Rôle
<code><c> ... </c></code>	Indiquer que le texte joint est un code d'application. La balise <code><code></code> est utilisée pour les lignes de code qui doivent être séparées de toute description jointe ; la balise <code><c></code> est utilisée pour le code incorporé à une description jointe.
<code><see cref="member"/></code>	Indiquer la référence à un autre membre ou champ. Le compilateur vérifie que ce « membre » existe réellement.
<code><seealso cref="member"/></code>	Indiquer la référence à un autre membre ou champ. Le compilateur vérifie que ce « membre » existe réellement. La différence entre les balises <code><see></code> et <code><seealso></code> 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.
<code><exception> ... </exception></code>	Fournir une description pour une classe d'exception.
<code><permission> ... </permission></code>	Documenter l'accessibilité d'un membre.
<code><param name="name"> ... </param></code>	Fournir une description pour un paramètre de méthode.
<code><returns> ... </returns></code>	Documenter la valeur renvoyée et le type de méthode.
<code><value> ... </value></code>	Décrire une propriété.

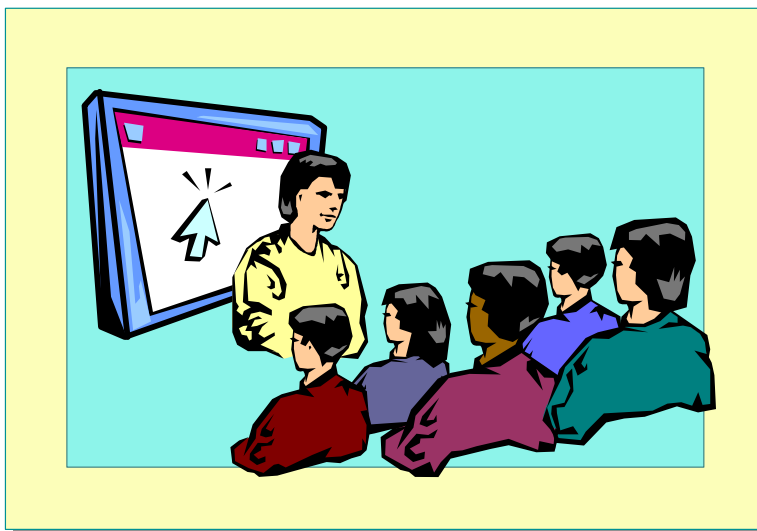
Vous pouvez compiler les balises XML et la documentation dans un fichier XML en utilisant le compilateur C# avec l'option `/doc` :

```
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.

Remarque 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.

Démonstration : Génération et affichage de la documentation XML



Dans cette démonstration, vous allez apprendre à compiler dans un fichier XML les commentaires XML incorporés à une application C#. Vous allez également apprendre à visualiser le fichier de documentation généré.

Gestion des exceptions

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

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.

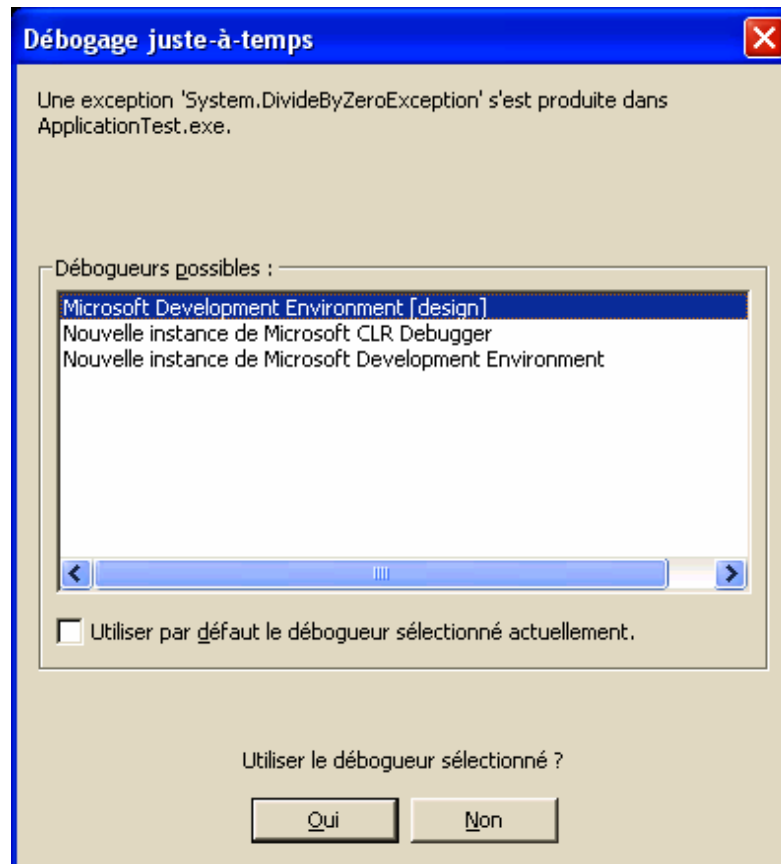
Si une erreur d'exécution se produit dans une application C#, le système d'exploitation renvoie une exception. Interception des exceptions à l'aide d'une construction **try-catch**, comme illustré sur la diapositive. Si l'une des instructions de la partie **try** de l'application provoque une exception,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

Remarque 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**.

Conseil Il est bien plus facile de définir la gestion des exceptions au démarrage de vos applications C# que d'essayer de le faire ultérieurement.

Si vous n'utilisez pas la gestion des exceptions, une exception d'exécution se produit. Si vous souhaitez déboguer votre programme à l'aide du débogage juste-à-temps, vous devez dans un premier temps l'activer. Si vous avez activé le débogage juste-à-temps, en fonction de l'environnement et des outils installés, le système de débogage juste-à-temps vous demande de définir le débogueur à utiliser.



Pour activer le débogage juste-à-temps, effectuez les étapes suivantes :

1. Dans le menu **Outils**, cliquez sur **Options**.
2. Dans la boîte de dialogue **Options**, cliquez sur le dossier **Débogage**.
3. Dans le dossier **Débogage**, cliquez sur **Juste-à-temps**.
4. Activez ou désactivez le débogage Juste-à-temps (JIT) pour des types de programmes spécifiques, puis cliquez sur **OK**.

Vous en apprendrez davantage sur le débogueur plus loin dans ce module.

◆ Compilation, exécution et débogage

- Appel du compilateur
- Exécution de l'application
- Démonstration : Compilation et exécution d'un programme C#
- Débogage
- Démonstration : Utilisation du débogueur Visual Studio
- Les outils du SDK
- Démonstration : Utilisation de ILDASM

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.

Appel du compilateur

- Principaux commutateurs de compilation
- Compilation à partir de la ligne de commande
- Compilation à partir de Visual Studio
- Localisation des erreurs

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.

Remarque 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.

Principaux commutateurs de compilation

Vous pouvez spécifier différents commutateurs pour le compilateur C# à l'aide de la commande **csc**. Le tableau suivant décrit les principaux commutateurs :

Commutateur	Signification
/?, /help	Affiche les options de compilation sur la sortie standard.
/out	Spécifie le nom de l'exécutable.
/main	Spécifie la classe contenant la méthode Main (si plusieurs classes de l'application incluent une méthode Main).
/optimize	Active ou désactive l'optimisateur de code.
/warn	Définit le niveau d'avertissement du compilateur.
/warnaserror	Traite tous les avertissements en tant qu'erreurs qui interrompent la compilation.
/target	Spécifie le type d'application générée.
/checked	Indique si le dépassement arithmétique génère une exception d'exécution.
/doc	Traite les commentaires de documentation pour créer un fichier XML.
/debug	Génère des informations de débogage.

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
```

Important 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**.

Remarque 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.

Vous pouvez modifier les options utilisées par le compilateur en mettant à jour la configuration du projet :

1. Dans l'Explorateur de solutions, cliquez avec le bouton droit sur l'icône du projet.
2. Cliquez sur **Propriétés**.
3. Dans la boîte de dialogue **Pages de propriétés**, cliquez sur **Propriétés de configuration**, puis sur **Générer**.
4. Spécifiez les options de compilation requises, puis cliquez sur **OK**.

Localisation des erreurs

Si le compilateur C# détecte des erreurs syntaxiques ou sémantiques, il les signale.

Si le compilateur a été appelé à partir de la ligne de commande, il affiche des messages indiquant les numéros de ligne et la position des caractères pour chaque ligne où une erreur a été détectée.

Si le compilateur a été appelé à partir de Visual Studio, la fenêtre Liste des tâches affiche toutes les lignes contenant des erreurs. Si vous double-cliquez sur chaque ligne de cette fenêtre, vous accédez aux erreurs correspondantes dans l'application.

Conseil Il est fréquent qu'une seule erreur de programmation génère plusieurs erreurs de compilation. Il est recommandé de corriger les erreurs en commençant par les premières erreurs qui ont été détectées, car la correction d'une erreur précoce peut corriger automatiquement plusieurs erreurs suivantes.

Exécution de l'application

- **Exécution à partir de la ligne de commande**
 - Tapez le nom de l'application
- **Exécution à partir de Visual Studio**
 - Cliquez sur **Exécuter sans débogage** dans le menu **Déboguer**

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

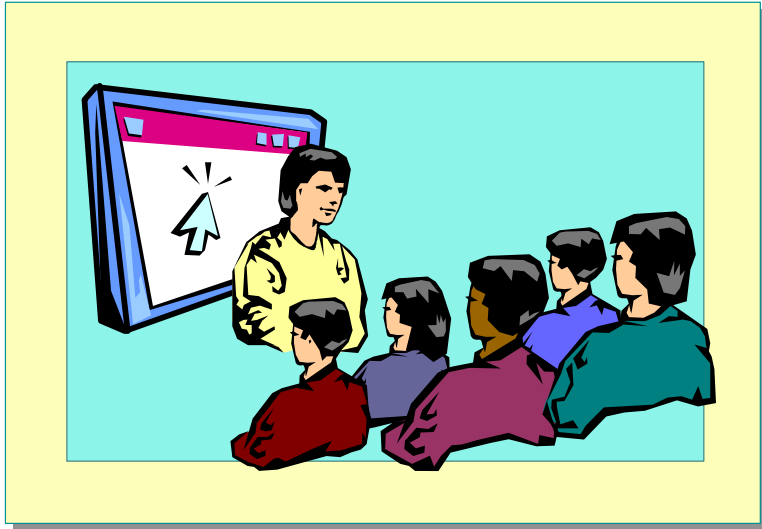
Exécution à partir de la ligne de commande

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.

Démonstration : Compilation et exécution d'un programme C#



Dans cette démonstration, vous allez apprendre à compiler et exécuter un programme C# à l'aide de Visual Studio. Vous allez également apprendre à repérer et corriger les erreurs de compilation.

Débogage

- **Exceptions et débogage JIT**
- **Le débogueur Visual Studio**
 - Définition de points d'arrêt et d'espions
 - Code pas à pas
 - Examen et modification des variables

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.

Remarque Le Kit de développement .NET Framework SDK contient un autre débogueur : `corDBG.exe`. Il s'agit d'un débogueur de ligne de commande. Il inclut la plupart des fonctionnalités offertes par Microsoft Development Environment, à l'exception de l'interface utilisateur graphique. Il ne sera pas abordé plus en détail dans ce cours.

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.

Important Pour utiliser le débogueur, vérifiez que vous avez sélectionné la configuration de solution Debug et non Release.

Code pas à pas

Après avoir défini tous les points d'arrêt nécessaires, vous pouvez exécuter votre application en cliquant sur **Démarrer** dans le menu **Déboguer** ou en appuyant sur F5. Lorsque le premier point d'arrêt est atteint, l'exécution s'interrompt.

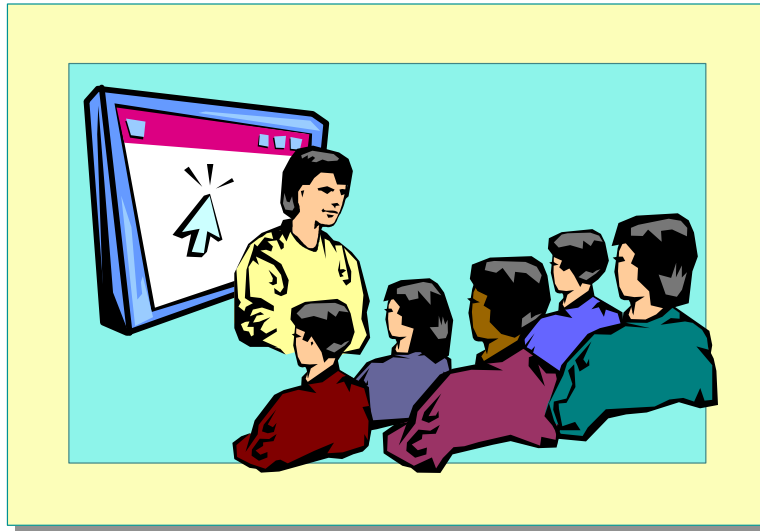
Vous pouvez continuer à exécuter l'application en cliquant sur **Continuer** dans le menu **Déboguer** ou bien utiliser l'une des options pas à pas du menu **Déboguer** pour parcourir votre code ligne par ligne.

Conseil Les options de point d'arrêt, de pas à pas et de surveillance des variables sont également disponibles dans la barre d'outils **Déboguer**.

Examen et modification des variables

Vous pouvez visualiser les variables définies dans la méthode en cours en cliquant sur **Variables locales** dans la barre d'outils **Déboguer** ou en utilisant la fenêtre Espion. Vous pouvez modifier par écrasement la valeur des variables (tout comme dans la fenêtre Espion).

Démonstration : Utilisation du débogueur Visual Studio



Cette démonstration va vous apprendre à utiliser le débogueur Visual Studio pour définir des points d'arrêt et des espions. Elle vous apprendra également à parcourir du code pas à pas ainsi qu'à examiner et modifier la valeur des variables.

Les outils du SDK

- Outils et utilitaires génériques
- Outils de conception et utilitaires Windows Forms
- Outils et utilitaires de sécurité
- Outils et utilitaires de configuration et de déploiement

Le Kit de développement .NET Framework SDK est fourni avec plusieurs outils qui offrent des fonctionnalités supplémentaires pour le développement, la configuration et le déploiement d'applications. Ces outils peuvent être exécutés à partir de la ligne de commande.

Outils et utilitaires génériques

Vous pouvez utiliser les outils génériques ci-dessous.

Nom de l'outil	Commande	Description
Runtime Debugger	cordbg.exe	Débogueur de ligne de commande.
MSIL Assembler	ilasm.exe	Assembleur qui utilise le langage MSIL comme entrée et génère un fichier exécutable.
MSIL Disassembler	ildasm.exe	Désassembleur pouvant être utilisé pour inspecter le langage MSIL et les métadonnées dans un fichier exécutable.
PEVerify	peverify.exe	Valide la sécurité du code et des métadonnées au niveau des types avant la diffusion.
Window Forms Class Viewer	wincv.exe	Localise les classes managées et affiche les informations les concernant.

Outils de conception et utilitaires Windows Forms

Vous pouvez utiliser les outils ci-dessous pour gérer et convertir les contrôles ActiveX® et Microsoft Windows® Forms.

Nom de l'outil	Commande	Description
Windows Forms ActiveX Control Importer	aximp.exe	L'outil ActiveX Control Importer convertit les définitions de types d'une bibliothèque de types COM pour un contrôle ActiveX en une commande Windows Forms.
License Compiler	lc.exe	L'outil License Compiler lit les fichiers texte qui contiennent des informations de licence et crée un fichier .licenses pouvant être intégré dans un exécutable Common Language Runtime en tant que ressource.
Utilitaire Resource File Generation	ResGen.exe	Crée un fichier .resources binaire pour code managé à partir de fichiers texte qui décrivent les ressources.
Windows Forms Resource Editor	winres.exe	L'outil Windows Forms Resource Editor est un outil de mise en page visuelle qui permet aux experts en localisation de localiser les formulaires Windows Forms. Les fichiers .resx ou .resources utilisés en tant qu'entrées du fichier Winres.exe sont généralement créés à l'aide d'un environnement de conception visuelle tel que Visual Studio .NET.

Outils et utilitaires de sécurité

Vous pouvez utiliser les outils ci-dessous pour fournir des fonctionnalités de sécurité et de cryptage pour les assemblys et les classes .NET.

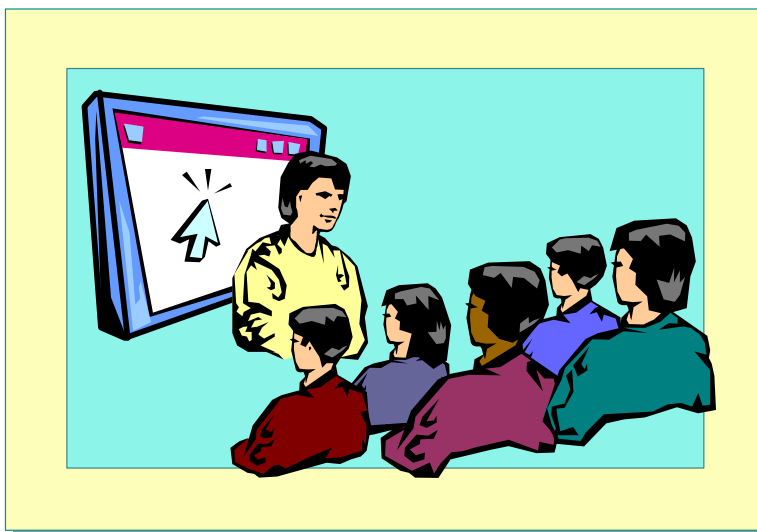
Nom de l'outil	Commande	Description
Code Access Security Policy	caspol.exe	L'outil Code Access Security Policy permet aux utilisateurs et aux administrateurs de modifier la stratégie de sécurité au niveau des ordinateurs, des utilisateurs et des entreprises.
Software Publisher Certificate Test	cert2spc.exe	Crée un certificat d'éditeur de logiciel (SPC, <i>Software Publisher's Certificate</i>) à partir d'un certificat X.509. Cet outil est uniquement utilisé pour effectuer des tâches de test.
Certificate Creation	makecert.exe	Version avancée de cert2spc.exe. Cet outil est uniquement utilisé pour effectuer des tâches de test.
Certificate Manager	certmgr.exe	Gère les certificats, les listes d'octroi de certificats et les listes de retrait de certificats.
Certificate Verification	chktrust.exe	Vérifie la validité d'un fichier signé.
Permissions View	permview.exe	Affiche les autorisations requises pour un assembly.
Secutil	SecUtil.exe	Localise la clé publique ou les informations de certificat dans un assembly.
Set Registry	setreg.exe	Modifie les paramètres du registre associés à la cryptographie à clé publique.
File Signing	signcode.exe	Signe un fichier exécutable ou un assembly à l'aide d'une signature numérique.
Strong Name	Sn.exe	Permet de créer des assemblys avec des noms forts. Cet outil garantit le caractère unique des noms et assure une certaine intégrité. Il permet également la signature des assemblys.

Outils et utilitaires de configuration et de déploiement

Une grande partie des outils ci-dessous sont des outils spécialisés que vous utiliserez uniquement si vous intégrez du code managé .NET Platform et des classes COM.

Nom de l'outil	Commande	Description
Assembly Generation	al.exe	Génère un manifeste de l'assembly à partir du langage MSIL et des fichiers de ressources.
Assembly Registration	RegAsm.exe	Permet aux classes managées .NET Platform d'être appelées de manière transparente par les composants COM.
Services Registration	RegSvcs.exe	Met les classes managées à disposition en tant que composants COM en chargeant et en enregistrant l'assembly et en générant et en installant une bibliothèque de types et une application COM+.
Assembly Cache Viewer	shfusion.dll	Affiche le contenu du cache global. Il s'agit d'une extension shell utilisée par l'Explorateur Microsoft Windows.
Isolated Storage	storeadm.exe	Gère le stockage isolé pour l'utilisateur actuellement connecté.
Type Library Exporter	TlbExp.exe	Convertit un assembly .NET en une bibliothèque de types COM.
Type Library Importer	Tlbimp.exe	Convertit les définitions de la bibliothèque de types COM au format de métadonnées équivalent utilisé par .NET.
XML Schema Definition	xsd.exe	L'outil XML Schema Definition génère des schémas XML ou des classes du Common Language Runtime à partir de fichiers XDR, XML et XSD ou de classes d'un assembly du Runtime.

Démonstration : Utilisation de ILDASM



Dans cette démonstration, vous allez apprendre à utiliser le langage MSIL (Microsoft Intermediate Language) pour examiner le manifeste et le code MSIL dans une classe.

Atelier 2.1 : Création d'un programme simple en C#



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer un programme en C# ;
- compiler et exécuter un programme en C# ;
- utiliser le débogueur Visual Studio ;
- ajouter la gestion des exceptions à un programme en C#.

Durée approximative de cet atelier : 60 minutes

Exercice 1

Création d'un programme simple en C#

Dans cet exercice, vous allez utiliser Visual Studio pour écrire un programme C#. Le programme vous demandera d'indiquer votre nom et vous accueillera ensuite par votre nom.

► Pour créer une nouvelle application console C#

1. Démarrez **Microsoft Visual Studio .NET**.
2. Dans le menu **Fichier**, pointez sur **Nouveau**, puis cliquez sur **Projet**.
3. Cliquez sur **Projets Visual C#** dans la zone **Types de projets**.
4. Cliquez sur **Application console** dans la zone **Modèles**.
5. Tapez **Greetings** dans la zone **Nom**.
6. Tapez *dossier d'installation*\Labs\Lab02 dans la zone **Emplacement** et cliquez sur **OK**.
7. Tapez le commentaire qui convient en guise de résumé.
8. Remplacez le nom de la classe par **Greeter**.
9. Enregistrez le projet en cliquant sur **Enregistrer tout** dans le menu **Fichier**.

► Pour écrire des instructions d'invite et d'accueil de l'utilisateur

1. Dans la méthode **Main**, après les commentaires TODO :, insérez la ligne suivante :
`string myName;`
2. Écrivez une instruction pour inviter les utilisateurs à indiquer leur nom.
3. Écrivez une autre instruction pour lire la réponse de l'utilisateur à partir du clavier et l'assigner à la chaîne *myName*.
4. Ajoutez une autre instruction pour imprimer « Bonjour *myName* » à l'écran (où *myName* est le nom saisi par l'utilisateur).
5. Une fois cette opération terminée, la méthode **Main** doit contenir les éléments suivants :

```
static void Main(string[ ] args)
{
    string myName;

    Console.WriteLine("Entrez votre nom");
    myName = Console.ReadLine( );
    Console.WriteLine("Bonjour {0}", myName);
}
```

6. Enregistrez votre travail

► **Pour compiler et exécuter le programme**

1. Dans le menu **Générer**, cliquez sur **Générer la solution** (ou appuyez sur CTRL+MAJ+B).
2. Corrigez toutes les erreurs de compilation et recommencez la génération si nécessaire.
3. Dans le menu **Déboguer**, cliquez sur **Exécuter sans débogage** (ou appuyez sur CTRL+F5).
4. Dans la fenêtre de console qui s'affiche, tapez votre nom lorsque le programme vous le demande et appuyez sur **ENTRÉE**.
5. Lorsque le message de bienvenue s'affiche, appuyez sur une touche à l'invite « Appuyez sur n'importe quelle touche pour continuer ».

Exercice 2

Compilation et exécution du programme C# à partir de la ligne de commande

Dans cet exercice, vous allez compiler et exécuter votre programme à partir de la ligne de commande.

► **Pour compiler et exécuter l'application à partir de la ligne de commande**

1. Cliquez sur le bouton **Démarrer** de Windows, pointez sur **Tous les programmes**, cliquez sur **Visual Studio .NET**, sur **Visual Studio .NET Tools**, puis sur **Invite de commandes Visual Studio .NET**.
2. Accédez au dossier *dossier d'installation*\Labs\Lab02\Greetings.
3. Compilez le programme à l'aide de la commande suivante :
`csc /out:Greet.exe Class1.cs`
4. Exécutez le programme en entrant la commande suivante :
`Greet`
5. Fermez la fenêtre Commande.

Exercice 3

Utilisation du débogueur

Dans cet exercice, vous allez utiliser le débogueur Visual Studio pour parcourir votre programme pas à pas et examiner la valeur d'une variable.

► Pour définir un point d'arrêt et commencer le débogage à l'aide de Visual Studio

1. Démarrez **Visual Studio .NET** si ce n'est déjà fait.
2. Dans le menu **Fichier**, pointez sur **Ouvrir**, puis cliquez sur **Projet**.
3. Ouvrez le projet Greetings.sln dans le dossier *dossier d'installation\Labs\Lab02\Greetings*.
4. Cliquez dans la marge gauche sur la ligne contenant la première occurrence de **Console.WriteLine** dans la classe **Greeter**.
Un point d'arrêt (gros point rouge) apparaît dans la marge.
5. Dans le menu **Déboguer**, cliquez sur **Démarrer** (ou appuyez sur F5).
L'exécution du programme commence, une fenêtre de console s'affiche, et le programme s'interrompt au point d'arrêt.

► Pour surveiller la valeur d'une variable

1. Dans le menu **Déboguer**, pointez sur **Fenêtres**, sur **Espion**, puis cliquez sur **Espion 1**.
2. Dans la fenêtre Espion, ajoutez la variable *myName* à la liste des variables surveillées.
3. La variable *myName* s'affiche dans la fenêtre Espion avec la valeur **null**.

► Pour parcourir le code pas à pas

1. Dans le menu **Déboguer**, cliquez sur **Pas à pas principal** (ou appuyez sur F10) pour exécuter la première instruction **Console.WriteLine**.
2. Passez à la ligne suivante contenant l'instruction **Console.ReadLine** en appuyant sur F10.
3. Retournez dans la fenêtre de console et tapez votre nom, puis appuyez sur la touche RETOUR.
Retournez dans Visual Studio. La valeur de *myName* dans la fenêtre Espion correspond à votre nom.
4. Passez à la ligne suivante contenant l'instruction **Console.WriteLine** en appuyant sur F10.
5. Affichez la fenêtre de console au premier plan.
Le message de bienvenue apparaît.
6. Retournez dans Visual Studio. Dans le menu **Déboguer**, cliquez sur **Continuer** (ou appuyez sur F5) pour exécuter le programme jusqu'à la fin.

Exercice 4

Ajout de la gestion des exceptions à un programme C#

Dans cet exercice, vous allez écrire un programme utilisant la gestion des exceptions pour détecter les erreurs d'exécution inattendues. Le programme demande à l'utilisateur de fournir deux valeurs entières. Il divise le premier nombre entier par le second et affiche le résultat.

► Pour créer un nouveau programme C#

1. Démarrez Visual Studio .NET si ce n'est déjà fait.
2. Dans le menu **Fichier**, pointez sur **Nouveau**, puis cliquez sur **Projet**.
3. Cliquez sur **Projets Visual C#** dans la zone **Types de projets**.
4. Cliquez sur **Application console** dans la zone **Modèles**.
5. Tapez **Divider** dans la zone **Nom**.
6. Tapez *dossier d'installation*\Labs\Lab02 dans la zone **Emplacement** et cliquez sur **OK**.
7. Tapez le commentaire qui convient en guise de résumé.
8. Remplacez le nom de la classe par **DivideIt**.
9. Enregistrez le projet en cliquant sur **Enregistrer tout** dans le menu **Fichier**.

► Pour écrire des instructions invitant l'utilisateur à fournir deux valeurs entières

1. Dans la méthode **Main**, écrivez une instruction invitant l'utilisateur à fournir la première valeur entière.
2. Écrivez une autre instruction pour lire la réponse de l'utilisateur à partir du clavier et l'assigner à une variable appelée *temp* de type **string**.
3. Ajoutez une instruction pour convertir la valeur de chaîne de la variable *temp* en un nombre entier et stocker le résultat dans *i* sous la forme suivante :

```
int i = Int32.Parse(temp);
```

4. Ajoutez des instructions à votre code pour :
 - a. Demander à l'utilisateur de fournir la seconde valeur entière.
 - b. Lire la réponse de l'utilisateur à partir du clavier et l'assigner à la variable *temp*.
 - c. Convertir la valeur de la variable *temp* en un nombre entier et stocker le résultat dans *j*.

Votre code doit ressembler à ce qui suit :

```
Console.WriteLine("Entrez la première valeur entière");  
string temp = Console.ReadLine();  
int i = Int32.Parse(temp);
```

```
Console.WriteLine("Entrez la seconde valeur entière");  
temp = Console.ReadLine();  
int j = Int32.Parse(temp);
```

5. Enregistrez votre travail.

► **Pour diviser le premier nombre entier par le second et afficher le résultat**

1. Écrivez du code pour créer une nouvelle variable de nombre entier k qui reçoit la valeur résultant de la division de i par j , et insérez cette variable à la fin de la procédure précédente. Votre code doit ressembler à ce qui suit :

```
int k = i / j;
```

2. Ajoutez une instruction pour afficher la valeur de k .
3. Enregistrez votre travail.

► **Pour tester le programme**

1. Dans le menu **Déboguer**, cliquez sur **Exécuter sans débogage** (ou appuyez sur CTRL+F5).
2. Tapez **10** comme première valeur entière et appuyez sur ENTRÉE.
3. Tapez **5** comme seconde valeur entière et appuyez sur ENTRÉE.
4. Vérifiez que la valeur affichée pour k est 2.
5. Exécutez le programme une nouvelle fois en appuyant sur CTRL+F5.
6. Tapez **10** comme première valeur entière et appuyez sur ENTRÉE.
7. Tapez **0** comme seconde valeur entière et appuyez sur ENTRÉE.
8. Le programme provoque le renvoi d'une exception (division par zéro).
9. Cliquez sur **Non** pour supprimer la boîte de dialogue Débogage Juste-à-temps.

► **Pour ajouter la gestion des exceptions au programme**

1. Placez le code dans la méthode **Main** au sein d'un bloc **try** de la manière suivante :

```
try
{
    Console.WriteLine (...);
    ...
    int k = i / j;
    Console.WriteLine (...);
}
```

2. Ajoutez une instruction **catch** à la méthode **Main**, après le bloc try. L'instruction **catch** doit afficher un message court, comme l'illustre le code suivant :

```
catch(Exception e)
{
    Console.WriteLine("Une exception a été levée :
↳{0}" , e);
}
...
```

3. Enregistrez votre travail.

4. La méthode **Main** complète doit ressembler à ce qui suit :

```
public static void Main(string[] args)
{
    try {
        Console.WriteLine("Entrez la première valeur
        ↪entière");
        string temp = Console.ReadLine();
        int i = Int32.Parse(temp);

        Console.WriteLine("Entrez la seconde valeur entière");
        temp = Console.ReadLine();
        int j = Int32.Parse(temp);

        int k = i / j;
        Console.WriteLine("Le résultat de la division de
        ↪{0} par {1} est {2}", i, j, k);
    }
    catch(Exception e) {
        Console.WriteLine("Une exception a été levée :
        ↪{0}" , e);
    }
}
```

► **Pour tester le code de gestion des exceptions**

1. Exécutez le programme une nouvelle fois en appuyant sur CTRL+F5.
2. Tapez **10** comme première valeur entière et appuyez sur ENTRÉE.
3. Tapez **0** comme seconde valeur entière et appuyez sur ENTRÉE.

Le programme provoque toujours le renvoi d'une exception (division par zéro), mais cette fois, l'erreur est interceptée et votre message s'affiche.

Contrôle des acquis

- Structure d'un programme en C#
- Opérations élémentaires d'entrée/sortie
- Pratiques recommandées
- Compilation, exécution et débogage

-
1. Quand commence l'exécution dans une application C# ?
 2. Quand se termine l'exécution d'une application ?
 3. Combien de classes une application C# peut-elle contenir ?
 4. Combien de méthodes **Main** une application peut-elle contenir ?

5. Comment lire l'entrée de l'utilisateur à partir du clavier dans une application C# ?
6. Dans quel espace de noms la classe **Console** se trouve-t-elle ?
7. Que se passe-t-il si votre application C# provoque la levée d'une exception qu'elle n'est pas prête à récupérer ?

Module 3 : Utilisation des variables de type valeur

Table des matières

Vue d'ensemble	1
Système de types communs (CTS, <i>Common Type System</i>)	2
Attribution de noms aux variables	8
Utilisation de types de données intégrés	14
Création de types de données définis par l'utilisateur	24
Conversion de types de données	28
Atelier 3.1 : Création et utilisation des types	32
Contrôle des acquis	36



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- **Système de types communs (CTS, *Common Type System*)**
- **Attribution de noms aux variables**
- **Utilisation de types de données intégrés**
- **Création de types de données définis par l'utilisateur**
- **Conversion de types de données**

Toutes les applications manipulent les données d'une certaine façon. En tant que développeur C#, il est impératif que vous sachiez stocker et traiter les données de vos applications. Lorsque votre application a besoin de stocker des données de façon temporaire en vue de les utiliser à l'exécution, vous pouvez les stocker dans une variable. Mais avant d'utiliser une variable, vous devez la définir. Pour cela, il faut lui réserver un espace de stockage en identifiant son type de données et en lui attribuant un nom. Une fois la variable définie, vous pouvez lui assigner des valeurs.

Dans ce module, vous allez apprendre à utiliser les variables de type valeur dans C#. Vous découvrirez comment spécifier le type de données contenu dans les variables, nommer les variables d'après les conventions d'affectation de noms standard et assigner des valeurs aux variables. Vous apprendrez également à modifier le type de données d'une variable existante et à créer vos propres variables.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- décrire les types de variables pouvant être utilisés dans les applications C# ;
- nommer vos variables en respectant les conventions d'affectation de noms C# standard ;
- déclarer des variables en utilisant des types de données intégrés ;
- attribuer des valeurs aux variables ;
- convertir des variables existantes d'un type de données à un autre ;
- créer et utiliser vos propres types de données.

◆ **Système de types communs (CTS, *Common Type System*)**

- **Vue d'ensemble du système de types communs**
- **Comparaison des types valeur et référence**
- **Comparaison des types valeur définis par l'utilisateur et des types valeur intégrés**
- **Types simples**

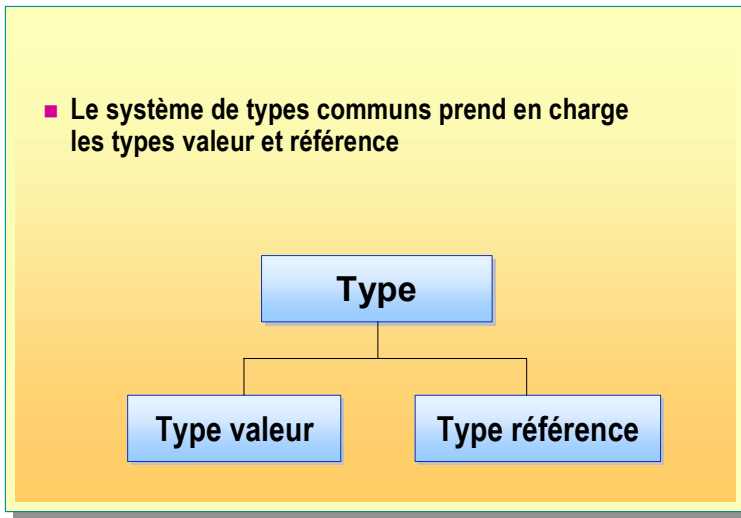
Chaque variable dispose d'un type de données qui détermine quelles valeurs elle peut stocker. C# est un langage sécurisé au niveau des types : le compilateur C# s'assure que les types des valeurs stockées dans les variables sont toujours appropriés.

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.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- décrire le fonctionnement du système de types communs ;
- choisir les types de données appropriés pour vos variables.

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.

Système de types communs

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.

Dans ce module, vous allez étudier deux types de variables.

- Variables de type valeur
- Variables de type référence

Comparaison des types valeur et référence

■ Variables de type valeur :	■ Variables de type référence :
■ Contiennent directement leurs données	■ Stockent des références à leurs données (connues comme objets)
■ Chacune dispose de son propre jeu de données	■ Deux variables référence peuvent référencer le même objet
■ Les opérations sur l'une n'affectent pas les autres	■ Les opérations sur l'une peuvent affecter les autres

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.

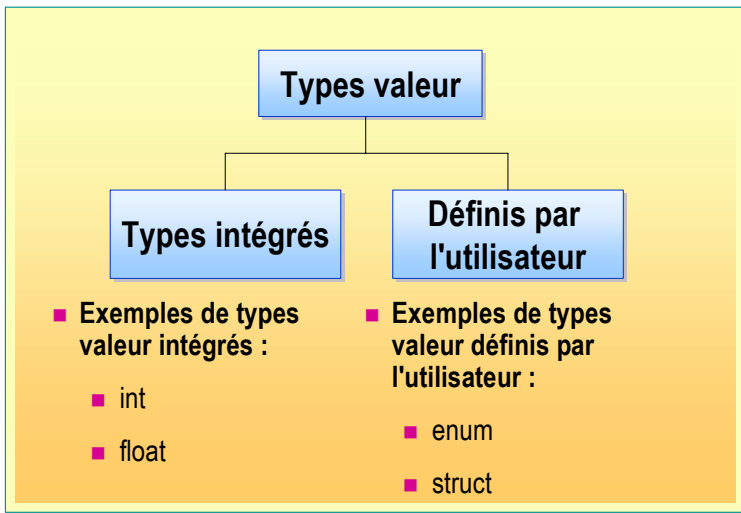
Types référence

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.

Remarque Tous les types de données de base sont définis dans l'espace de noms **System**. Tous les types sont dérivés de **System.Object**. Les types valeur sont dérivés de **System.ValueType**.

Pour plus d'informations sur les types référence, consultez le module 8, « Utilisation des variables de type référence » du cours 2132A, *Programmation en C#*.

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**.

Dans ce module, vous allez apprendre à créer des types de données définis par l'utilisateur, tels que les types structure ou énumération.

Types simples

- **Identifiés à l'aide de mots clés réservés**

- `int` // mot clé réservé

-ou-

- `System.Int32`

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 réservé

– ou –

`System.Byte` // type struct

`int` // mot clé réservé

– ou –

`System.Int32` // type struct

Pour plus d'informations sur les tailles et les plages des types valeur intégrés, faites une recherche sur « Types valeur » dans l'aide de Microsoft® Visual Studio® .NET.

Le tableau suivant 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.UInt64
Char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

◆ Attribution de noms aux variables

- Règles et recommandations pour l'affectation de noms aux variables
- Mots clés en C#
- Quiz : Pouvez-vous repérer les noms de variables non autorisés ?

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.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- identifier les mots clés réservés standard en C# ;
- nommer vos variables en respectant les conventions d'affectation de noms C# standard.

Règles et recommandations pour l'affectation de noms aux variables

■ **Règles**

- utiliser des lettres, le trait de soulignement et des chiffres

■ **Recommandations**

- éviter d'utiliser uniquement des 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 (PascalCasing) dans les noms composés, à savoir une majuscule à l'initiale de chaque mot.

Réponse42
42Réponse

✓

✗

different
Different

✓

✓

MAUVAISSTYLE
_styledéconseillé
MeilleurStyle

✗

✗

✓

Msg
Message

✗

✓

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 (PascalCasing) dans les noms composés, à savoir une majuscule à l'initiale de chaque mot.

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;
```

Pour plus d'informations sur les conventions d'affectation de noms, consultez la section « Indications concernant l'attribution d'un nom » dans l'aide du Kit de développement Microsoft .NET Framework SDK (*Software Development Kit*, Kit de développement de logiciel).

Mots clés en C#

- Les mots clés sont des identificateurs réservés

```
abstract, base, bool, default, if, finally
```

- N'utilisez pas de mot clé comme nom de variable

- Provoque une erreur de compilation

- Évitez d'utiliser des mots clés en modifiant leur casse

```
int INT; // Style déconseillé
```

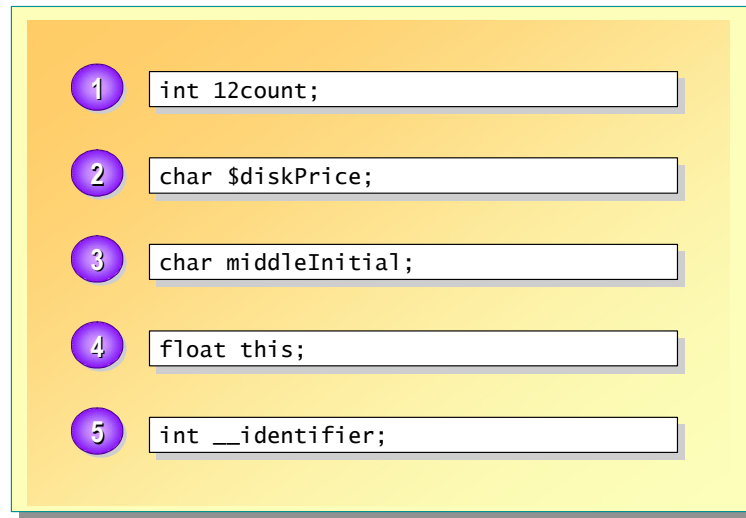
Les mots clés sont réservés, ce qui signifie que vous ne pouvez pas les utiliser comme noms de variables en C#. L'utilisation d'un mot clé comme nom de variable entraînera une erreur de compilation.

Mots clés de C#

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			

Quiz : Pouvez-vous repérer les noms de variables non autorisés ?



1

2

3

4

5

Réponses au quiz

1. **Interdit.** Les noms de variables ne peuvent pas commencer par un chiffre.
2. **Interdit.** Les noms de variables doivent commencer par une lettre ou un trait de soulignement.
3. **Autorisé.** Les noms de variables peuvent commencer par une lettre.
4. **Interdit.** Les mots clés (**this**) ne peuvent pas être utilisés pour nommer les variables.
5. **Autorisé.** Les noms de variables peuvent commencer par un trait de soulignement.

◆ Utilisation de types de données intégrés

- Déclaration de variables locales
- Attribution de valeurs aux variables
- Assignment composée
- Opérateurs courants
- Incrémentation et décrémentation
- Ordre de priorité des opérateurs

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.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- créer une variable locale en utilisant des types de données intégrés ;
- attribuer des valeurs aux variables à l'aide d'opérateurs ;
- définir des variables et des constantes en lecture seule.

Déclaration de variables locales

- Généralement déclarées à l'aide du type de données et du nom de la variable :

```
int itemCount;
```

- Possibilité de déclarer plusieurs variables dans une seule déclaration :

```
int itemCount, employeeNumber;
```

-OU-

```
int itemCount,  
    employeeNumber;
```

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);
```

Attribution de valeurs aux variables

- **Assignez des valeurs aux variables déjà déclarées :**

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

- **Initialisez une variable à sa déclaration :**

```
int employeeNumber = 23;
```

- **Vous pouvez initialiser des valeurs de type caractère :**

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

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';
```

Assignment composée

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

```
itemCount = itemCount + 40;
```

- La syntaxe abrégée est pratique

```
itemCount += 40;
```

- Cette syntaxe abrégée fonctionne pour tous les opérateurs arithmétiques

```
itemCount -= 24;
```

Ajouter une valeur à une variable est une opération très courante

Le code suivant déclare une variable **int** appelée *itemCount*, lui attribue la valeur 2, puis l'incrémente de 40 :

```
int itemCount;  
itemCount = 2;  
itemCount = itemCount + 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
```

Opérateurs courants

Opérateurs courants	Exemple
• Opérateurs d'égalité	== !=
• Opérateurs relationnels	< > <= >= is
• Opérateurs conditionnels	&& ?:
• Opérateur d'incrément	++
• Opérateur de décrément	--
• Opérateurs arithmétiques	+ - * / %
• Opérateurs d'assignation	= *= /= %= += -= <<= >>= &= ^= =

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

La concaténation et l'opérateur d'addition (+), l'opérateur de soustraction (-), l'opérateur de multiplication (*) ainsi que l'opérateur de division (/) sont des exemples d'opérateurs. Les littéraux, les champs, les variables locales et les expressions sont des exemples d'opérandes.

Opérateurs courants

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

Type	Description
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	Compèrent deux valeurs.
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.

Pour plus d'informations sur les opérateurs disponibles dans C#, consultez « Expressions » dans les spécifications du langage C# dans l'aide de Visual Studio .NET.

Incrémentation et décrémentation

- **Modifier une valeur d'une unité est une opération très courante**

```
itemCount += 1;  
itemCount -= 1;
```

- **La syntaxe abrégée est pratique**

```
itemCount++;  
itemCount--;
```

- **Cette syntaxe abrégée existe sous deux formes**

```
++itemCount;  
--itemCount;
```

Modifier une valeur d'une unité est une opération très courante

Il vous arrive souvent d'écrire des instructions qui incrémentent ou décrémentent une valeur d'une unité. Pour ce faire, vous pouvez procéder comme suit :

```
itemCount = itemCount + 1;  
itemCount = itemCount - 1;
```

Toutefois, comme expliqué précédemment, il existe une syntaxe abrégée plus pratique à cet effet :

```
itemCount += 1;  
itemCount -= 1;
```

Syntaxe abrégée pratique

Incrémenter ou décrémenter une valeur d'une unité est une opération si courante qu'il existe une forme encore plus abrégée de la syntaxe !

```
itemCount++; // itemCount += 1;  
itemCount--; // itemCount -= 1;
```

L'opérateur ++ est appelé opérateur d'incrément et l'opérateur -- est appelé opérateur de décrémentation. Vous pouvez considérer ++ comme un opérateur qui incrémente une valeur et -- comme un opérateur qui décrémenté une valeur.

Comme déjà indiqué, cette syntaxe abrégée est la forme idiomatique préférée des programmeurs C# pour incrémenter ou décrémenter une valeur d'une unité.

Remarque Le langage C++ est appelé ainsi car il succède à C !

Cette syntaxe abrégée existe sous deux formes

Vous pouvez utiliser les opérateurs ++ et -- sous deux formes.

1. Le symbole de l'opérateur peut être placé *avant* l'identificateur, comme l'illustrent les exemples suivants. Il s'agit de la notation *préfixée*.

```
++itemCount;  
--itemCount;
```

2. Le symbole de l'opérateur peut être placé *après* l'identificateur, comme l'illustrent les exemples suivants. Il s'agit alors de la notation *postfixée*.

```
itemCount++;  
itemCount--;
```

Dans les deux cas, la variable *itemCount* est incrémentée (pour ++) ou décrémentée (pour --) d'une unité. Pourquoi existe-t-il deux notations ? Pour répondre à cette question, vous devez mieux comprendre les mécanismes de l'assignation :

L'une des fonctionnalités importantes de C# est que l'assignation est un opérateur. En d'autres termes, une expression d'assignation ne se contente pas d'attribuer une valeur à une variable ; elle a elle-même une valeur, ou résultat, qui est la valeur de la variable une fois que l'assignation a été effectuée. Dans la plupart des instructions, la valeur de l'expression d'assignation est ignorée, mais elle peut être utilisée dans une expression plus longue, comme dans l'exemple suivant :

```
int itemCount = 0;  
Console.WriteLine(itemCount = 2); // Affiche 2  
Console.WriteLine(itemCount = itemCount + 40); // Affiche 42
```

L'assignation composée est également une assignation. En d'autres termes, une expression d'assignation composée ne se contente pas d'attribuer une valeur à une variable ; elle a elle-même une valeur, ou résultat. Là encore, dans la plupart des instructions, la valeur de l'expression d'assignation composée est ignorée, mais elle peut être utilisée dans une expression plus longue, comme dans l'exemple suivant :

```
int itemCount = 0;  
Console.WriteLine(itemCount += 2); // Affiche 2  
Console.WriteLine(itemCount -= 2); // Affiche 0
```

L'incréméntation et la décrémentation sont aussi des assignations. Cela signifie, par exemple, qu'une expression d'incréméntation, outre qu'elle incrémente une variable d'une unité, a aussi elle-même une valeur, un résultat. Là encore, dans la plupart des instructions, la valeur de l'expression d'incréméntation est ignorée, mais elle peut être utilisée aussi dans une expression plus longue, comme dans l'exemple suivant :

```
int itemCount = 42;
int prefixValue = ++itemCount; // prefixValue == 43
int postfixValue = itemCount++; // postfixValue = 43
```

La valeur de l'expression d'incréméntation varie selon que vous utilisez ou non la version préfixée ou postfixée. Dans les deux cas, *itemCount* est incrémenté. Le problème ne se situe pas à ce niveau. Ce qui est important, c'est la valeur de l'expression d'incréméntation. La valeur de pré-incrémentation/pré-décréméntation est la valeur de la variable *après* l'incréméntation/la décrémentation. La valeur de post-incrémentation/post-décréméntation est la valeur de la variable *avant* l'incréméntation/la décrémentation.

Ordre de priorité des opérateurs

■ Ordre de priorité des opérateurs et interaction

- À l'exception des opérateurs d'assignation, tous les opérateurs binaires sont associatifs à gauche
- Les opérateurs d'assignation et conditionnels sont associatifs à droite

Ordre de priorité des opérateurs

Lorsqu'une expression contient plusieurs opérateurs, l'ordre de priorité des opérateurs contrôle l'ordre dans lequel les opérateurs individuels sont évalués. Par exemple, l'expression $x + y * z$ est évaluée sous la forme $x + (y * z)$, car l'opérateur de multiplication a une priorité supérieure à celle de l'opérateur d'addition. Par exemple, une *expression additive* se compose d'une suite d'*expressions multiplicatives* séparées par des opérateurs $+$ ou $-$, ce qui confère aux opérateurs $+$ et $-$ une priorité plus basse que celle des opérateurs $*$, $/$ et $\%$.

Interaction

Lorsqu'un opérande est placé entre deux opérateurs de même priorité, l'*interaction* des opérateurs régit l'ordre dans lequel les opérations sont effectuées. Par exemple, $x + y + z$ est évalué sous la forme $(x + y) + z$. C'est particulièrement important pour les opérateurs d'assignation. Par exemple, $x = y = z$ est évalué sous la forme $x = (y = z)$.

- À l'exception des opérateurs d'assignation, tous les opérateurs binaires sont *associatifs à gauche*, ce qui veut dire que les opérations sont effectuées de gauche à droite.
- Les opérateurs d'assignation et l'opérateur conditionnel ($?:$) sont *associatifs à droite*, ce qui veut dire que les opérations sont effectuées de droite à gauche.

Vous pouvez contrôler la priorité et l'interaction à l'aide de parenthèses. Par exemple, $x + y * z$ commence par multiplier y par z , puis additionne le résultat à x , alors que $(x + y) * z$ additionne tout d'abord x et y , puis multiplie le résultat par z .

◆ Création de types de données définis par l'utilisateur

- Types énumération (**enum**)
- Types structure (**struct**)

Dans vos applications, vous devez savoir comment créer des types de données **énumération** (**enum**) et **structure** (**struct**) définis par l'utilisateur.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- créer des types de données **enum** définis par l'utilisateur ;
- créer des types de données **struct** définis par l'utilisateur.

Types énumération (enum)

■ Définition d'un type énumération

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

■ Utilisation d'un type énumération

```
Color colorPalette = Color.Red;
```

■ Affichage d'une variable énumération

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

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.

Vous pouvez initialiser une énumération en spécifiant des littéraux entiers.

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);
```

Types structure (struct)

■ Définition d'un type structure

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

■ Utilisation d'un type structure

```
Employee companyEmployee;
companyEmployee.firstName = "Jean";
companyEmployee.age = 23;
```

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;
```

◆ Conversion de types de données

- Conversion implicite de types de données
- Conversion explicite de types de données

Dans C#, il existe deux types de conversion de données :

- la conversion implicite de types de données ;
- la conversion explicite de types de données.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- effectuer une conversion implicite de données ;
- effectuer une conversion explicite de données.

Conversion implicite de types de données

- Pour convertir un type **int** en type **long** :

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

- Les conversions implicites ne peuvent pas échouer

- Perte de précision possible, mais pas des informations

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);
    }
}
```

Conversion explicite de types de données

- Pour effectuer des conversions explicites, utilisez 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);
    }
}
```

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);
    }
}
```

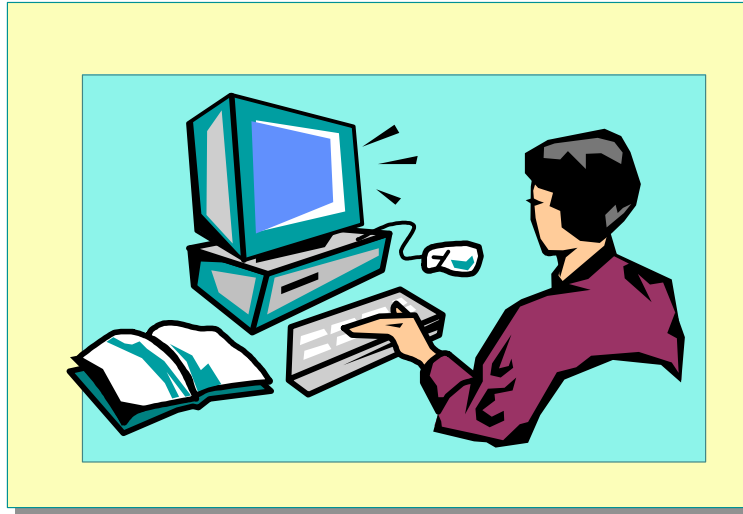
Un dépassement s'étant produit dans cet exemple, le résultat est le suivant :

(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);
        }
    }
}
```

Atelier 3.1 : Création et utilisation des types



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer de nouveaux types de données ;
- définir et utiliser des variables.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les éléments suivants :

- système de types communs (CTS, *Common Type System*) ;
- variables de type valeur dans C#.

Scénario

Dans l'exercice 1, vous écrirez un programme qui crée un type simple **enum**, puis définit et affiche les valeurs en utilisant l'instruction **Console.WriteLine**.

Dans l'exercice 2, vous écrirez un programme qui utilise le type **enum** déclaré dans l'exercice 1 dans un **struct**.

S'il vous reste du temps, vous ajouterez des fonctionnalités d'entrée/sortie au programme que vous avez écrit dans l'exercice 2.

Fichiers de démarrage et de solution

Cet atelier comporte plusieurs fichiers de démarrage et de solution. Les fichiers de démarrage se trouvent dans le dossier *dossier d'installation\Labs\Lab03\Starter* et les fichiers de solution, dans le dossier *dossier d'installation\Labs\Lab03\Solution*.

Durée approximative de cet atelier : 35 minutes

Exercice 1

Création d'un type enum

Dans cet exercice, vous allez créer un type énuméré pour représenter différents types de comptes bancaires (courant et épargne). Vous créerez deux variables en utilisant le type **enum** et affecter aux variables les valeurs Courant et Epargne. Vous afficherez ensuite les valeurs des variables en utilisant la fonction **System.Console.WriteLine**.

► Pour créer un type enum

1. Ouvrez le projet BankAccount.sln dans le dossier *dossier d'installation*\Labs\Lab03\Starter\BankAccount.
2. Ouvrez le fichier Enum.cs et ajoutez un **enum** appelé **AccountType** avant la définition de la classe comme suit :

```
public enum AccountType { Courant, Epargne }
```

Cet **enum** contiendra des types Courant et Epargne.

3. Déclarez deux variables de type **AccountType** dans **Main** comme suit :

```
AccountType goldAccount;  
AccountType platinumAccount;
```

4. Affectez à la première variable la valeur Courant et à l'autre variable la valeur Epargne comme suit :

```
goldAccount = AccountType.Courant;  
platinumAccount = AccountType.Epargne;
```

5. Ajoutez deux instructions **Console.WriteLine** pour imprimer la valeur de chaque variable comme suit :

```
Console.WriteLine("Le type de compte client est {0}"  
    ↪,goldAccount);  
Console.WriteLine("Le type de compte client est {0}"  
    ↪,platinumAccount);
```

6. Compilez et exécutez le programme.

Exercice 2

Création et utilisation d'un type struct

Dans cet exercice, vous allez définir un type **struct** qui peut être utilisé pour représenter un compte bancaire. Vous utiliserez des variables pour contenir le numéro du compte (**long**), le solde du compte (**decimal**) et le type du compte (**enum**, que vous avez créé dans l'exercice 1). Vous créerez une variable de type **struct**, remplirez le **struct** avec des exemples de données et imprimerez le résultat.

► Pour créer un type struct

1. Ouvrez le projet StructType.sln dans le dossier *dossier d'installation*\Labs\Lab03\Starter\StructType.
2. Ouvrez le fichier Struct.cs et ajoutez un **struct de type public** appelé **BankAccount**, qui contient les champs suivants.

Type	Variable
public long	<i>accNo</i>
public decimal	<i>accBal</i>
public AccountType	<i>accType</i>

3. Déclarez une variable *goldAccount* de type **BankAccount** dans **Main**.

```
BankAccount goldAccount;
```

4. Définissez les champs *accType*, *accBal* et *accNo* de la variable *goldAccount*.

```
goldAccount.accType = AccountType.Courant;  
goldAccount.accBal = (decimal)3200.00;  
goldAccount.accNo = 123;
```

5. Ajoutez les instructions **Console.WriteLine** pour imprimer la valeur de chaque élément de la variable struct.

```
Console.WriteLine("Numéro de compte {0}",  
    ↪goldAccount.accNo);  
Console.WriteLine("Type de compte {0}",  
    ↪goldAccount.accType);  
Console.WriteLine("Solde du compte {0}  
    ↪Euros",goldAccount.accBal);
```

6. Compilez et exécutez le programme.

S'il vous reste du temps

Ajout de fonctionnalités d'entrée/sortie

Dans cet exercice, vous allez modifier le code écrit dans l'exercice 2. Au lieu d'utiliser le numéro de compte 123, vous inviterez l'utilisateur à entrer le numéro de compte. Vous utiliserez ce numéro pour imprimer la synthèse du compte.

► Pour ajouter des fonctionnalités d'entrée/sortie

1. Ouvrez le projet StructType.sln dans le dossier *dossier d'installation*\Labs\Lab03\Starter\Optional.
2. Ouvrez le fichier Struct.cs et remplacez la ligne suivante :
`goldAccount.accNo = 123; //supprimer cette ligne et ajouter le code ci-dessous`

par une instruction `Console.Write` pour inviter l'utilisateur à entrer le numéro de compte :

```
Console.Write("Entrer un numéro de compte : ");
```

3. Lisez le numéro de compte en utilisant l'instruction **Console.ReadLine**. Affectez cette valeur à *goldAccount.accNo*.
`goldAccount.accNo = long.Parse(Console.ReadLine());`

Remarque Vous devez utiliser la méthode **long.Parse** pour convertir la chaîne lue par l'instruction **Console.ReadLine** en valeur décimale avant de l'affecter à *goldAccount.accNo*.

4. Compilez et exécutez le programme. Entrez un numéro de compte lorsque vous y serez invité.

Contrôle des acquis

- Système de types communs (CTS, *Common Type System*)
- Attribution de noms aux variables
- Utilisation de types de données intégrés
- Création de types de données définis par l'utilisateur
- Conversion de types de données

-
1. Qu'est-ce que le système de types communs ?
 2. Un type valeur peut-il être **null** ?
 3. Pouvez-vous utiliser des variables non initialisées dans C# ? Pourquoi ?
 4. Une conversion implicite peut-elle entraîner une perte d'informations ?

Module 4 : Instructions et exceptions

Table des matières

Vue d'ensemble	1
Introduction aux instructions	2
Utilisation des instructions conditionnelles	6
Utilisation des instructions d'itération	17
Utilisation des instructions de saut	29
Atelier 4.1 : Utilisation des instructions	32
Gestion des exceptions fondamentales	42
Levée d'exceptions	52
Atelier 4.2 : Utilisation des exceptions	62
Contrôle des acquis	72



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Introduction aux instructions
- Utilisation des instructions conditionnelles
- Utilisation des instructions d'itération
- Utilisation des instructions de saut
- Gestion des exceptions fondamentales
- Levée d'exceptions

L'aptitude à écrire dans un langage particulier les instructions qui constituent la logique d'un programme est l'une des compétences essentielles d'un programmeur. Ce module décrit le fonctionnement et la syntaxe de quelques instructions courantes en C#, ainsi que la mise en œuvre de la gestion des exceptions dans le langage C#.

Ce module explique en particulier comment lever des erreurs et les intercepter, ainsi que l'utilisation des blocs **try-finally** pour s'assurer qu'une exception n'entraîne pas l'arrêt du programme sans « nettoyage » préalable.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- décrire les différents types d'instructions de contrôle ;
- utiliser les instructions de saut ;
- utiliser les instructions conditionnelles ;
- utiliser les instructions d'itération ;
- gérer et lever des exceptions.

◆ Introduction aux instructions

- Blocs d'instructions
- Types d'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.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- grouper une suite d'instructions en C# ;
- utiliser les différents types d'instructions disponibles en C#.

Blocs d'instructions

The diagram consists of a yellow rectangular background containing three separate white boxes, each with a title and a code snippet. The first box on the left is titled 'Utilisez des accolades comme délimiteurs de blocs' and shows a simple block with a comment. The middle box is titled 'Un bloc et son bloc parent ne peuvent pas contenir une variable du même nom' and shows a nested block structure where the inner block declares a variable with the same name as one in the outer block. The third box on the right is titled 'Des blocs frères peuvent contenir des variables du même nom' and shows two sibling blocks at the same indentation level, each declaring a variable with the same name.

- Utilisez des accolades comme délimiteurs de blocs

```
{  
    // code  
}
```

- Un bloc et son bloc parent ne peuvent pas contenir une variable du même nom

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

- Des blocs frères peuvent contenir des variables du même nom

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

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 }. Vous n'utilisez pas de délimiteurs de mots clés correspondants, tels que **If ... End If** de Microsoft® Visual Basic®.

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 :

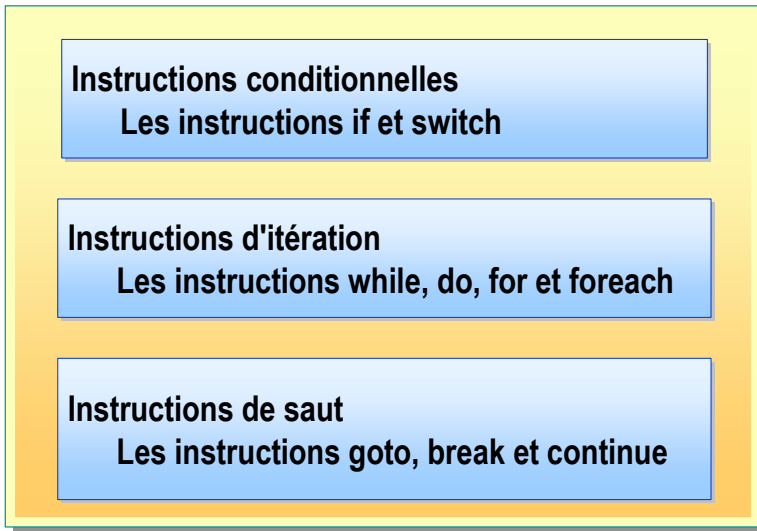
```
int i;
{
    int i; // Erreur : i déjà déclaré dans le bloc parent
    ...
}
```

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.

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.

◆ Utilisation des instructions conditionnelles

- L'instruction **if**
- Instructions **if** en cascade
- L'instruction **switch**
- Quiz : Trouver les bogues

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.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- utiliser l'instruction **if** en C# ;
- utiliser l'instruction **switch** en C#.

L'instruction if

■ Syntaxe :

```
if ( expression-bol  enne )  
    premi  re-instruction-incorpor  e  
else  
    seconde-instruction-incorpor  e
```

■ Absence de conversion implicite du type int au type bool

```
int x;  
...  
if (x) ...    // Doit   tre if (x != 0) en C#  
if (x = 0) ... // Doit   tre if (x == 0) en C#
```

La premi  re instruction d  cisionnelle est l'instruction **if**. Elle peut   tre associ  e    une clause **else** facultative, comme illustr   ci-dessous :

```
if ( expression-bol  enne )  
    premi  re-instruction-incorpor  e  
else  
    seconde-instruction-incorpor  e
```

L'instruction **if**   value une expression bool  enne afin de d  terminer la marche    suivre. Si l'expression bool  enne a la valeur **true**, la premi  re instruction incorpor  e est ex  cut  e. Si l'expression bool  enne a la valeur **false** et qu'il y a une clause **else**, le contr  le est pass      la seconde instruction incorpor  e.

Exemples

Vous pouvez utiliser une simple instruction **if** incorporée comme dans le code suivant :

```
if (number % 2 == 0)
    Console.WriteLine("paire");
```

Bien que les accolades ne soient pas obligatoires dans les instructions incorporées, de nombreux guides de style recommandent leur utilisation parce qu'elles évitent les erreurs de code et facilitent la maintenance. Vous pouvez réécrire l'exemple précédent avec des accolades comme suit :

```
if (number % 2 == 0) {
    Console.WriteLine("paire");
}
```

Vous pouvez également utiliser un bloc d'instructions **if**, tel que celui-ci :

```
if (minute == 60) {
    minute = 0;
    hour++;
}
```

Conversion de valeurs de type entier en valeurs de type booléen

La conversion implicite d'une valeur de type entier en valeur de type booléen représentant une source potentielle de bogues, C# ne gère pas ce type de conversion. C'est l'une des différences notables entre C# et d'autres langages de la même famille.

Ainsi, les instructions suivantes qui, au mieux, génèrent des avertissements en C et C++, produisent des erreurs de compilation dans C# :

```
int x;
...
if (x) ... // Doit être x != 0 en C#
if (x = 0) ... // Doit être x == 0 en C#
```


Instructions if en cascade

```
enum Suite {Trèfle, Coeur, Carreau, Pique}  
Suite atout = Suite.Coeur;  
if (atout == Suite.Trèfle)  
    couleur = "Noir";  
else if (atout == Suite.Coeur)  
    couleur = "Rouge";  
else if (atout == Suite.Carreau)  
    couleur = "Rouge";  
else  
    couleur = "Noir";
```

L'instruction **else if** permet de traiter les instructions **if** en cascade. C# ne prend pas en charge l'instruction **else if**, mais permet de construire une instruction de type **else if** avec une clause **else** et une instruction **if**, tout comme dans les langages C et C++. Des langages tels que Visual Basic gèrent les instructions **if** en cascade en utilisant une instruction **else if** entre l'instruction **if** de début et l'instruction **else** de fin.

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.

Imbrication d'instructions if

L'imbrication d'une instruction **if** dans une autre est susceptible de créer une ambiguïté de type *else non résolu*, comme illustré dans l'exemple ci-dessous :

```
if (percent >= 0 && percent <= 100)
    if (percent > 50)
        Console.WriteLine("Succès");
else
    Console.WriteLine("Erreur : hors limite");
```

L'instruction **else** et la première instruction **if** sont au même niveau d'indentation. En lisant ce code, on pourrait penser que la clause **else** n'est pas associée au second **if**, ce qui est tout à fait faux. Quelle que soit la présentation du code, le compilateur associe la clause **else** avec la plus proche instruction **if** qui la précède. Autrement dit, le compilateur interprétera l'exemple de code ci-dessus de la manière suivante :

```
if (percent >= 0 && percent <= 100) {
    if (percent > 50)
        Console.WriteLine("Succès");
    else
        Console.WriteLine("Erreur : hors limite");
}
```

Pour associer la clause **else** avec le premier **if**, vous pouvez utiliser un bloc, comme suit :

```
if (percent >= 0 && percent <= 100) {
    if (percent > 50)
        Console.WriteLine("Succès");
} else {
    Console.WriteLine("Erreur : hors limite");
}
```

Conseil L'indentation correcte des instructions **if** en cascade améliore la lisibilité du programme et évite d'empiéter sur la marge droite de la page ou de l'écran.

L'instruction switch

- Utilisez les instructions **switch** pour plusieurs blocs **case**
- Utilisez les instructions **break** pour empêcher tout passage

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

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. Vous ne pouvez pas regrouper une collection de constantes dans une seule étiquette **case** ; chaque constante doit posséder sa propre étiquette **case**.

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**.

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.

Remarque Contrairement aux langages Java, C ou C++, le type gouvernant une instruction **switch** en C# peut être une chaîne. Dans le cas d'une expression de type chaîne, la valeur **null** est autorisée comme constante d'une étiquette **case**.

Pour plus d'informations sur les opérateurs de conversion, recherchez « opérateurs de conversion » dans l'aide du Microsoft .NET Framework SDK.

Regroupement de constantes

Pour regrouper plusieurs constantes, répétez le mot clé **case** devant chaque constante, comme illustré dans l'exemple ci-dessous :

```
enum MonthName { January, February, ..., December }
MonthName current;
int monthDays;
...
switch (current) {
case MonthName.February :
    monthDays = 28;
    break;
case MonthName.April :
case MonthName.June :
case MonthName.September :
case MonthName.November :
    monthDays = 30;
    break;
default :
    monthDays = 31;
    break;
}
```

Les étiquettes **case** et **default** servent uniquement de points d'entrée pour le flux de contrôle du programme, selon la valeur de l'expression **switch** ; elles ne modifient pas le flux de contrôle du programme.

Les valeurs des constantes **case** doivent être uniques ; autrement dit, deux constantes ne peuvent pas avoir la même valeur. Ainsi, l'exemple suivant génère une erreur de compilation :

```
switch (atout) {
case Suite.Trèfle :
case Suite.Trèfle : // Erreur : étiquette en double
    ...
default :
default : // Erreur : étiquette en double
    ...
}
```

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.

L'exemple de code suivant, qui enfreint la règle de non-passage, génère une erreur de compilation :

```
string suffix = "th";
switch (days % 10) {
case 1 :
    if (days / 10 != 1) {
        suffix = "st";
        break;
    }
    // Erreur : passage ici
case 2 :
    if (days / 10 != 1) {
        suffix = "nd";
        break;
    }
    // Erreur : passage ici
case 3 :
    if (days / 10 != 1) {
        suffix = "rd";
        break;
    }
    // Erreur : passage ici
default :
    suffix = "th";
    // Erreur : passage ici
}
```

Vous pouvez corriger cette erreur en réécrivant le code comme suit :

```
switch (days % 10) {
case 1 :
    suffix = (days / 10 == 1) ? "th" : "st";
    break;
case 2 :
    suffix = (days / 10 == 1) ? "th" : "nd";
    break;
case 3 :
    suffix = (days / 10 == 1) ? "th" : "rd";
    break;
default :
    suffix = "th";
    break;
}
```

Utilisation de l'instruction goto dans les instructions switch

Dans C#, contrairement aux langages Java, C ou C++, l'instruction **goto** permet de se déplacer sur une étiquette **case** ou **default**. Au besoin, vous pouvez recourir à une instruction **goto** de cette manière pour passer d'une étiquette à une autre. Ainsi, l'exemple de code suivant sera compilé sans erreur :

```
switch (days % 10) {
case 1 :
    if (days / 10 != 1) {
        suffix = "st";
        break;
    }
    goto case 2;
case 2 :
    if (days / 10 != 1) {
        suffix = "nd";
        break;
    }
    goto case 3;
case 3 :
    if (days / 10 != 1) {
        suffix = "rd";
        break;
    }
    goto default;
default :
    suffix = "th";
    break;
}
```

La règle de non-passage permet de réorganiser les sections d'une instruction **switch** sans influencer sur son comportement global.

Quiz : Trouver les bogues

```
if number % 2 == 0    ...
```

1

```
if (percent < 0) || (percent > 100) ...
```

2

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

3

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

4

Vous pouvez travailler avec un partenaire pour trouver les bogues du code reproduit sur la diapositive. Les réponses se trouvent à la page suivante.

Réponses :

1. L'instruction **if** n'est pas entre parenthèses. Le compilateur C# intercepte ce bogue en tant qu'erreur de compilation. Le code correct est le suivant :
2. La totalité de l'instruction **if** n'est pas entre parenthèses. Le compilateur C# intercepte ce bogue en tant qu'erreur de compilation. Le code correct est le suivant :

```
if (number % 2 == 0) ...
```

```
if ((percent < 0) || (percent > 100)) ...
```

3. Une instruction incorporée de l'instruction **if** comporte uniquement un point-virgule. Un point-virgule est appelé instruction vide dans le document de référence du langage C# ou instruction **null** dans les messages de diagnostic du compilateur C#. Cette instruction ne fait rien, mais elle est autorisée. La présentation de ces instructions n'a pas d'incidence sur la manière dont le compilateur analyse la syntaxe du code. Le compilateur lit donc le code comme suit :

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

Le compilateur C# intercepte ce bogue en tant qu'avertissement de compilation.

4. Le code comporte les erreurs suivantes :
 - a. Une étiquette **case** comporte plusieurs constantes. Le compilateur C# intercepte ce bogue en tant qu'erreur de compilation.
 - b. Les instructions associées à chaque **case** passent à l'étiquette **case** suivante. Le compilateur C# intercepte ce bogue en tant qu'erreur de compilation.
 - c. L'orthographe du mot clé **default** est incorrecte. Ce code est malheureusement autorisé puisqu'il crée une simple étiquette d'identification. Le compilateur C# intercepte ce bogue sous la forme de deux avertissements de compilation : l'un signale l'impossibilité d'atteindre une partie du code ; l'autre indique que l'étiquette **default** n'a pas été utilisée.

◆ Utilisation des instructions d'itération

- L'instruction **while**
- L'instruction **do**
- L'instruction **for**
- L'instruction **foreach**
- Quiz : Trouver les bogues

Le langage C# dispose de quatre instructions de boucle (également appelées instructions d'itération) : **while**, **do**, **for** et **foreach**. Elles sont utilisées pour effectuer des opérations tant qu'une condition particulière est vraie.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- utiliser les instructions d'itération dans C# ;
- identifier les erreurs dans les instructions d'itération en C#.

L'instruction while

- Exécute des instructions incorporées en fonction d'une valeur booléenne
- Évalue une expression booléenne au début de la boucle
- Exécute les instructions incorporées tant que la valeur booléenne est vraie

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

0 1 2 3 4 5 6 7 8 9

Parmi les instructions d'itération, **while** est la plus simple à utiliser. Elle exécute une instruction incorporée *tant que* l'expression booléenne est vraie. Remarquez que l'expression que l'instruction **while** évalue doit être booléenne puisque C# ne prend pas en charge la conversion implicite d'un entier vers une valeur booléenne.

Déroulement de l'exécution

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.

Exemples

Si une seule instruction dépend de **while**, vous pouvez utiliser la syntaxe suivante :

```
while (i < 10)
    Console.WriteLine(i++);
```

Vous n'êtes pas tenu de mettre les instructions incorporées entre accolades. De nombreux guides de style recommandent néanmoins leur utilisation pour simplifier la maintenance. Vous pouvez réécrire l'exemple précédent avec des accolades comme suit :

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

Vous pouvez également utiliser un bloc d'instructions **while** :

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

Conseil Si **while** est l'instruction d'itération la plus simple, elle peut toutefois poser des problèmes en cas d'inattention. Voici la syntaxe classique d'une instruction **while** :

```
initialiseur
while ( expression-boléenne ) {
    instruction-incorporée
    actualisation
}
```

Il est facile d'oublier la partie *actualisation* d'un bloc **while**, surtout lorsqu'on se concentre sur l'expression booléenne.

L'instruction do

- Exécute des instructions incorporées en fonction d'une valeur booléenne
- Évalue une expression booléenne à la fin de la boucle
- Exécute les instructions incorporées tant que la valeur booléenne est vraie

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

0 1 2 3 4 5 6 7 8 9

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 fois ou plus.

Une instruction **do** exécute donc toujours son instruction incorporée au moins une fois. Ce comportement est utile pour valider une entrée avant d'autoriser le programme à poursuivre son exécution.

Déroulement de l'exécution

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**.

Exemples

Si une seule instruction dépend de **do**, vous pouvez utiliser la syntaxe suivante :

```
do
    Console.WriteLine(i++);
while (i < 10);
```

Tout comme pour les instructions **if** et **while**, les accolades ne sont pas obligatoires autour des instructions incorporées de **do**, mais il est recommandé de les utiliser.

Vous pouvez également utiliser un bloc d'instructions **do**, tel que celui-ci :

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

Dans tous les cas, vous devez terminer une instruction **do** par un point-virgule :

```
do {
    Console.WriteLine(i++);
} while (i < 10) // Erreur : absence de ;
```

L'instruction for

- On place les informations d'actualisation au début de la boucle

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

0 1 2 3 4 5 6 7 8 9

- Les variables d'un bloc for sont limitées à ce bloc

```
for (int i = 0; i < 10; i++)  
    Console.WriteLine(i);  
Console.WriteLine(i); // Erreur : i n'est plus dans la portée
```

- Une boucle for peut parcourir par itération plusieurs valeurs

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

Les développeurs oublient souvent d'actualiser la variable de contrôle dans les instructions **while**. Voici un exemple de ce type d'erreur.

```
int i = 0;  
while (i < 10)  
    Console.WriteLine(i); // Faute : absence de i++
```

Cette erreur se produit parce que le développeur concentre son attention sur le corps de l'instruction **while**, et non sur l'actualisation. Par ailleurs, le mot clé **while** et le code d'actualisation sont souvent éloignés l'un de l'autre.

L'instruction **for** permet de minimiser ce type d'erreur. Dans cette instruction, 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
```

Important Dans une instruction **for**, le code d'actualisation précède l'instruction incorporée, mais il est exécuté par le runtime après l'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.

Exemples

Les composantes initialiseur, condition et actualisation d'une instruction **for** sont facultatives. Il convient toutefois de noter qu'une condition vide étant implicitement considérée **true**, cela peut générer une boucle infinie. En voici un exemple :

```
for (;;) {
    Console.WriteLine("Aide ");
    ...
}
```

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);

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

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. Cette règle s'applique également aux variables déclarées dans le code d'initialisation d'une instruction **for**. 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
```

Le code suivant est néanmoins autorisé :

```
for (int i = 0; i < 10; i++) ...  
for (int i = 0; i < 20; i++) ...
```

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; ... ; ...)
```

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++, 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.

L'instruction foreach

- Permet de choisir le type et le nom de la variable d'itération
- Exécute les instructions incorporées pour chaque élément de la classe de collection

```
ArrayList numbers = new ArrayList( );  
for (int i = 0; i < 10; i++) {  
    numbers.Add(i);  
}  
  
foreach (int number in numbers) {  
    Console.WriteLine(number);  
}
```

0 1 2 3 4 5 6 7 8 9

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 par exemple le code suivant :

```
using System.Collections;  
...  
ArrayList numbers = new ArrayList( );  
for (int i = 0; i < 10; i++) {  
    numbers.Add(i);  
}
```

Vous pouvez écrire une instruction **for** qui accède à chacun des éléments de cette classe de collection, et qui les affiche :

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

Cette instruction **for** comporte plusieurs instructions distinctes qui, ensemble, mettent en œuvre le mécanisme employé pour parcourir chaque élément (nombre) de la collection. Non seulement cette solution n'est pas simple à implémenter, mais elle présente des risques d'erreur.

Pour résoudre ce problème, 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.

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

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

L'instruction **foreach** exécute l'instruction incorporée pour chaque élément de la classe de collection *numbers*. Il suffit de choisir le type et le nom de la variable d'itération qui, dans cet exemple, sont respectivement **int** et *number*.

L'instruction **foreach** ne permet pas de modifier les éléments d'une collection parce que la variable d'itération est implicitement **readonly**. Par exemple :

```
foreach (int number in numbers) {
    number++; // Erreur de compilation
    Console.WriteLine(number);
}
```

Conseil Pour itérer sur les valeurs d'un énumérateur avec une instruction **foreach**, vous devez faire appel à la méthode **Enum.GetValues()** qui retourne un tableau d'objets.

Il convient de choisir avec soin le type de la variable d'itération **foreach**. Dans certaines circonstances, un type de variable d'itération erroné n'est décelé qu'à l'exécution.

Quiz : Trouver les bogues

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

1

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

2

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

3

```
do  
    ...  
    string line = Console.ReadLine( );  
    guess = int.Parse(line);  
while (guess != answer);
```

4

Vous pouvez travailler avec un partenaire pour trouver les bogues du code reproduit sur la diapositive. Les réponses se trouvent à la page suivante.

Réponses :

1. Les éléments de l'instruction **for** sont séparés par des virgules, et non par des points-virgules. Le compilateur C# intercepte ce bogue en tant qu'erreur de compilation. Le code correct est le suivant :

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

2. L'instruction **while** n'actualise pas l'expression d'itération. La boucle ne se terminera jamais. Ce bogue ne génère ni avertissement ni erreur de compilation. Le code correct est le suivant :

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

3. L'instruction **for** comporte une condition de fin au lieu d'une condition d'itération. Il n'y aura jamais de boucle. Ce bogue ne génère ni avertissement ni erreur de compilation. Le code correct est le suivant :

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

4. Les instructions entre **do** et **while** doivent être placées dans un bloc. Le compilateur C# intercepte ce bogue en tant qu'erreur de compilation. Le code correct est le suivant :

```
do {  
    ...  
    string line = Console.ReadLine( );  
    guess = int.Parse(line);  
} while (guess != answer);
```

◆ Utilisation des instructions de saut

- L'instruction **goto**
- Les instructions **break** et **continue**

Les instructions **goto**, **break** et **continue** sont des instructions de saut. Elles permettent de déplacer le contrôle d'exécution d'un point à un autre du programme, à tout moment. Dans cette section, vous allez apprendre à utiliser les instructions de saut dans les programmes en C#.

L'instruction goto

- Contrôle d'exécution transféré à une instruction marquée par une étiquette
- Peut facilement générer du code incompréhensible

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

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**.

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

Les instructions break et continue

- L'instruction **break** sort d'une itération
- L'instruction **continue** passe à l'itération suivante

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

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. Par exemple, vous pouvez réécrire l'instruction **while** de la diapositive avec **break** ou **continue**, comme suit :

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

Il est préférable de réécrire ce code avec une instruction **for** :

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

Atelier 4.1: Utilisation des instructions



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- contrôler le déroulement de l'exécution à l'aide d'instructions ;
- utiliser les instructions de bouclage.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les éléments suivants :

- création de variables en C# ;
- utilisation des opérateurs courants en C# ;
- création de types **enum** en C#.

Durée approximative de cet atelier : 30 minutes

Exercice 1

Conversion d'un jour de l'année en paire jour / mois

Cet exercice consiste à écrire un programme qui lit sur la console un nombre entier représentant le jour de l'année (compris entre 1 et 365), puis le stocke dans une variable de type entier. Le programme convertira ce numéro pour le mois et pour le jour du mois, puis affichera le résultat sur la console. Par exemple, en entrant 40, « 9 février » doit s'afficher. (Pour simplifier, les années bissextiles sont ignorées dans cet exercice.)

► Pour lire le numéro du jour sur la console

1. Ouvrez le projet WhatDay1.sln situé dans *dossier d'installation*\Labs\Lab04\Starter\WhatDay1. La classe **WhatDay** comporte une variable contenant le nombre de jours dans chaque mois, stockée dans une collection. Il n'est pas nécessaire, pour l'instant, de comprendre le fonctionnement de cette classe.
2. Ajoutez à **WhatDay.Main** une instruction **System.Console.Write** invitant l'utilisateur à entrer un numéro de jour compris entre 1 et 365.
3. Ajoutez à **Main** une instruction qui déclare une variable **string** nommée *line*, et initialisez-la à partir de la ligne lue sur la console avec la méthode **System.Console.ReadLine**.
4. Ajoutez à **Main** une instruction qui déclare une variable **int** nommée *dayNum*, et initialisez-la avec l'entier renvoyé par la méthode **int.Parse**.

Voici le code complet :

```
using System;

class WhatDay
{
    static void Main( )
    {
        Console.Write("Entrez un nombre compris entre
↪1 et 365 : ");
        string line = Console.ReadLine( );
        int dayNum = int.Parse(line);

        //
        // TODO : Ajouter le code ici
        //
    }
    ...
}
```

5. Enregistrez votre travail.
6. Compilez le programme WhatDay1.cs et corrigez les erreurs, le cas échéant. Exécutez le programme.

► **Pour calculer une paire jour / mois à partir du nombre correspondant à un jour**

1. Ajoutez à **Main** une instruction qui déclare une variable **int** nommée *monthNum*, et initialisez-la à zéro.
2. Dix instructions **if** commentées (une par mois, de « janvier » à « octobre ») sont fournies. Supprimez les commentaires des dix instructions **if**, et ajoutez à **Main** deux instructions **if** similaires pour les mois de « novembre » et « décembre ».

Conseil Pour supprimer plusieurs commentaires (en conservant leur contenu), sélectionnez les lignes de commentaire, puis choisissez **Edition**, **Options avancées** et **Ne pas commenter la sélection**.

3. Ajoutez à **Main** une étiquette nommée **End** sous la dernière instruction **if**.
4. Sous l'étiquette **End**, ajoutez une instruction qui déclare une variable **string** non initialisée, nommée *monthName*.
5. Une instruction **switch** est partiellement fournie après l'étiquette **End**. Sont également fournies dix étiquettes **case** commentées pour les mois de « janvier » à « octobre ». Supprimez ces commentaires, et ajoutez deux instructions **case** similaires avec leur contenu pour les mois de « novembre » et « décembre ». Ajoutez à l'instruction **switch** une étiquette **default**. Ajoutez à l'étiquette **default** une instruction affectant le littéral **string** « pas encore terminé » à la variable *monthName*.
6. Après l'instruction **switch**, utilisez la méthode **WriteLine** pour obtenir les valeurs de *dayNum* et *monthName*.

7. Voici le programme complet :

```
using System;

class WhatDay
{
    static void Main( )
    {
        Console.Write("Entrez un nombre compris entre
↪1 et 365 : ");
        string line = Console.ReadLine( );
        int dayNum = int.Parse(line);

        int monthNum = 0;

        if (dayNum <= 31) { // janvier
            goto End;
        } else {
            dayNum -= 31;
            monthNum++;
        }

        if (dayNum <= 28) { // février
            goto End;
        } else {
            dayNum -= 28;
            monthNum++;
        }

        if (dayNum <= 31) { // mars
            goto End;
        } else {
            dayNum -= 31;
            monthNum++;
        }
    }
}
```

Suite du code à la page suivante.

```
    if (dayNum <= 30) { // avril
        goto End;
    } else {
        dayNum -= 30;
        monthNum++;
    }

    if (dayNum <= 31) { // mai
        goto End;
    } else {
        dayNum -= 31;
        monthNum++;
    }

    if (dayNum <= 30) { // juin
        goto End;
    } else {
        dayNum -= 30;
        monthNum++;
    }

    if (dayNum <= 31) { // juillet
        goto End;
    } else {
        dayNum -= 31;
        monthNum++;
    }

    if (dayNum <= 31) { // août
        goto End;
    } else {
        dayNum -= 31;
        monthNum++;
    }

    if (dayNum <= 30) { // septembre
        goto End;
    } else {
        dayNum -= 30;
        monthNum++;
    }

    if (dayNum <= 31) { // octobre
        goto End;
    } else {
        dayNum -= 31;
        monthNum++;
    }

    if (dayNum <= 30) { // novembre
        goto End;
    } else {
        dayNum -= 30;
        monthNum++;
    }
}
```

Suite du code à la page suivante.

```
        if (dayNum <= 31) { // décembre
            goto End;
        } else {
            dayNum -= 31;
            monthNum++;
        }

    End:
    string monthName;

    switch (monthNum) {
    case 0 :
        monthName = "janvier"; break;
    case 1 :
        monthName = "février"; break;
    case 2 :
        monthName = "mars"; break;
    case 3 :
        monthName = "avril"; break;
    case 4 :
        monthName = "mai"; break;
    case 5 :
        monthName = "juin"; break;
    case 6 :
        monthName = "juillet"; break;
    case 7 :
        monthName = "août"; break;
    case 8 :
        monthName = "septembre"; break;
    case 9 :
        monthName = "octobre"; break;
    case 10 :
        monthName = "novembre"; break;
    case 11 :
        monthName = "décembre"; break;
    default :
        monthName = "pas encore terminé"; break;
    }

    Console.WriteLine("{0} {1}", dayNum, monthName);
}
...
}
```

8. Enregistrez votre travail.

9. Compilez le programme WhatDay1.cs et corrigez les erreurs, le cas échéant. Exécutez le programme. Vérifiez le bon fonctionnement du programme à l'aide des données suivantes.

Numéro de jour	Jour et mois
32	1 février
60	1 mars
91	1 avril
186	5 juillet
304	31 octobre
309	5 novembre
327	23 novembre
359	25 décembre

► **Pour calculer le nom du mois avec un énumérateur**

1. Vous remplacerez maintenant l'instruction **switch** qui détermine le nom du mois à partir d'un numéro de mois, par un mécanisme plus compact. Déclarez un type **enum** nommé **MonthName**, et renseignez-le avec les noms des douze mois de l'année, en commençant par « janvier » et en terminant avec « décembre ».
2. Commentez l'intégralité de l'instruction **switch**.

Conseil Pour commenter plusieurs lignes de code, sélectionnez les lignes, puis choisissez Edition, Options avancées et Commenter la sélection.

3. Remplacez l'instruction **switch** par une instruction qui déclare une variable **enum MonthName** nommée *temp*. Initialisez *temp* à partir de la variable **int monthNum**. Vous aurez besoin de cette expression cast :
`(MonthName)monthNum`
4. Remplacez l'initialisation de *monthName* par l'expression
`temp.ToString()`

5. Voici le programme complet :

```
using System;

enum MonthName
{
    janvier,
    février,
    mars,
    avril,
    mai,
    juin,
    juillet,
    août,
    septembre,
    octobre,
    novembre,
    décembre
}

class WhatDay
{
    static void Main( )
    {
        Console.Write("Entrez un nombre compris entre
↪ 1 et 365 : ");
        string line = Console.ReadLine( );
        int dayNum = int.Parse(line);

        int monthNum = 0;

        // 12 instructions if, comme ci-dessus

        End:

        MonthName temp = (MonthName)monthNum;
        string monthName = temp.ToString( );

        Console.WriteLine("{0} {1}", dayNum, monthName);
    }
    ...
}
```

6. Enregistrez votre travail.
7. Compilez le programme WhatDay1.cs et corrigez les erreurs, le cas échéant. Exécutez le programme. Testez les données du tableau précédent pour vérifier que le programme fonctionne toujours correctement.

► **Pour remplacer les 12 instructions if par une instruction foreach**

1. Vous remplacerez maintenant les 12 instructions qui calculent les paires jour / mois par une instruction **foreach**. Commentez les 12 instructions **if**. Vous les remplacerez par la suite.
2. Écrivez une instruction **foreach** qui parcourt la collection **DaysInMonths** fournie. Pour ce faire, ajoutez l'instruction suivante :
`foreach (int daysInMonth in DaysInMonths) ...`
3. Ajoutez une instruction de bloc qui constituera le corps de l'instruction **foreach**. Le contenu de ce bloc sera très similaire à une instruction **if** unique commentée, mais vous utiliserez la variable *daysInMonth* à la place d'une liste de littéraux de type entier.
4. Commentez l'étiquette **End** qui se trouve au-dessus de l'instruction **switch**. Dans l'instruction **foreach**, remplacez l'instruction **goto** par une instruction **break**.
5. Voici le programme complet :

```
using System;

enum MonthName { ... }

class WhatDay
{
    static void Main( )
    {
        Console.Write("Entrez un nombre compris entre
↪ 1 et 365 : ");
        string line = Console.ReadLine( );
        int dayNum = int.Parse(line);

        int monthNum = 0;

        foreach (int daysInMonth in DaysInMonths) {
            if (dayNum <= daysInMonth)
            {
                break;
            }
            else
            {
                dayNum -= daysInMonth;
                monthNum++;
            }
        }
        MonthName temp = (MonthName)monthNum;
        string monthName = temp.ToString( );

        Console.WriteLine("{0} {1}", dayNum, monthName);
    }
    ...
}
```


6. Enregistrez votre travail.
7. Compilez le programme WhatDay1.cs et corrigez les erreurs, le cas échéant. Exécutez le programme. Testez les données du tableau précédent pour vérifier que le programme fonctionne toujours correctement.
8. Exécutez le programme en entrant des numéros de jour inférieurs à 1 et supérieurs à 365 pour tester le résultat.

◆ Gestion des exceptions fondamentales

- Pourquoi utiliser des exceptions ?
- Objets exception
- Utilisation des blocs try et catch
- Blocs catch multiples

Un développeur a souvent l'impression de passer plus de temps sur la vérification et la gestion des erreurs que sur la logique du programme. Les exceptions, qui sont conçues pour traiter les erreurs, apportent une solution à ce problème. Dans cette section, vous allez apprendre à intercepter et à gérer les exceptions en C#.

Pourquoi utiliser des exceptions ?

- La gestion des erreurs traditionnelle basée sur les procédures est fastidieuse

```
int errorCode = 0;
FileInfo source = new FileInfo("code.cs");
if (errorCode == -1) goto Failed;
int length = (int)source.Length;
if (errorCode == -2) goto Failed;
char[] contents = new char[length];
if (errorCode == -3) goto Failed;
// Succès ...
Failed: ...
```

Logique essentielle
du programme

Gestion des
erreurs

Un programme solide est capable de prévoir l'imprévisible et d'y répondre intelligemment s'il se produit. Des erreurs peuvent survenir à tout moment, ou presque, pendant la compilation et l'exécution d'un programme.

La logique essentielle du programme reproduit sur la diapositive est la suivante :

```
FileInfo source = new FileInfo("code.cs");
int length = (int)source.Length;
char[ ] contents = new char[length];
...
```

Ces instructions essentielles sont malheureusement noyées dans une masse confuse de code de gestion des erreurs, qui obscurcit la logique du programme de plusieurs manières :

- La logique du programme et le code de gestion des erreurs sont présentés pêle-mêle.

Mêlées à du code de gestion des erreurs, les instructions essentielles du programme perdent leur intégrité conceptuelle. Le programme est difficile à lire et à comprendre.

- Tout le code d'erreur se ressemble.

Toutes les instructions de contrôle des erreurs sont semblables. Elles testent toutes le même code d'erreur avec des instructions **if**, et beaucoup de code est dupliqué, ce qui est toujours mauvais signe.

- La signification intrinsèque des codes d'erreur n'est pas claire.

Dans ce code, le sens de `-1`, par exemple, n'est pas explicite ; il peut signifier « Erreur de sécurité : lecture non autorisée », mais seule la documentation peut révéler sa signification. Les erreurs de code sous la forme d'un entier ne décrivent pas les erreurs qu'ils représentent.

- Les codes d'erreur sont définis au niveau des méthodes.

Chaque méthode consigne ses erreurs en définissant le code d'erreur sur une valeur unique qui lui est propre. Une même valeur ne peut pas être utilisée par deux méthodes différentes. Cela signifie que chaque méthode doit être liée à chacune des autres méthodes. Le code suivant, dans lequel une énumération remplace les codes d'erreur représentés par un entier, illustre bien ce lien :

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

Ce code est plus clair : Un identificateur, tel que `FileNotFound` est évidemment plus descriptif que `-1`. Néanmoins, l'ajout à **enum** d'un nouvel identificateur d'erreur aura une incidence sur toutes les méthodes qui nomment leurs erreurs dans cet **enum**, et nécessitera une recompilation.

- Le pouvoir descriptif d'un entier est très limité.

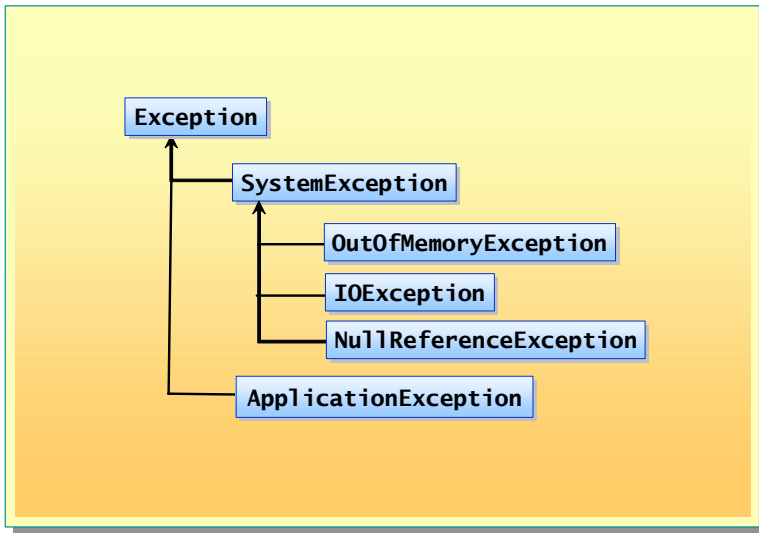
Par exemple, `-1` peut être documenté comme signifiant « Erreur de sécurité : lecture non autorisée », mais `-1` ne peut pas également fournir le nom du fichier faisant l'objet de cette restriction.

- Les codes d'erreur sont trop facilement ignorés.

Par exemple, les programmeurs en C ne vérifient presque jamais la valeur **int** renvoyée par la fonction **printf**. Celle-ci échoue rarement, mais dans ce cas, elle renvoie un entier négatif (généralement `-1`).

Il est clair que la méthode traditionnelle de gestion des erreurs n'est pas idéale. Les exceptions offrent une solution plus souple, elles sont plus efficaces et produisent des messages d'erreur significatifs.

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. La diapositive présente la hiérarchie de ces 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.

Conseil Il convient de trouver un équilibre entre des classes d'exceptions trop vagues et des classes d'exceptions trop précises. Dans le premier cas, vous ne pourrez pas écrire de bloc catch utile. En revanche, si la classe d'exceptions est trop précise, vous risquez de divulguer des détails d'implémentation et de rompre l'encapsulation.

L'exemple de code de la diapositive 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** (lequel, sur la diapositive, écrit le message contenu dans l'objet exception **OverflowException**).

Avec les blocs **try** et **catch**, les instructions de gestion des erreurs et les instructions logiques principales ne sont plus mêlées, et le programme est beaucoup plus clair.

Blocs catch multiples

- Chaque bloc **catch** intercepte une classe d'exceptions
- Un bloc **try** peut comporter un bloc **catch** général
- Un bloc **try** n'est pas autorisé à intercepter une classe dérivée d'une classe interceptée dans un bloc **catch** antérieur

```
try
{
    Console.WriteLine("Entrez le premier nombre");
    int i = int.Parse(Console.ReadLine());
    Console.WriteLine("Entrez un second nombre");
    int j = int.Parse(Console.ReadLine());
    int k = i / j;
}
catch (OverflowException caught) {...}
catch (DivideByZeroException caught) {...}
```

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. }
16. ...
```

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. Le contrôle exécute cette instruction jusqu'à son accolade fermante, puis repasse à la ligne 16.

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, puis est transféré sur la ligne 16.

Si aucune des instructions du bloc **try** ne lève d'exception, le flux de contrôle atteint la fin de ce bloc et passe à la ligne 16. 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**.

Si le flux de contrôle échoue dans sa recherche d'un bloc **catch** approprié, il arrête l'appel de méthode en cours et reprend sa recherche au niveau de l'instruction où l'appel de méthode avait été lancé. Il continue sa recherche, en déroulant la pile des appels jusqu'à **Main** s'il le faut. Si cela provoque l'arrêt de la méthode **Main**, le thread ou processus ayant appelé **Main** est arrêté selon la procédure définie par l'implémentation.

Bloc catch général

Le bloc **catch** général, également appelé clause **catch** générale, peut intercepter toute exception, quelle que soit sa classe ; il est souvent utilisé pour intercepter les exceptions qui, faute de gestionnaire approprié, risqueraient de ne pas être traitées.

Deux possibilités s'offrent à vous pour écrire un bloc **catch** général : Vous pouvez écrire une simple instruction **catch**, telle que celle-ci :

```
catch { ... }
```

Ou vous pouvez l'écrire sous cette forme :

```
catch (System.Exception) { ... }
```

Un bloc **try** ne peut comporter qu'un seul bloc **catch** général. Ainsi, l'exemple suivant génère une erreur :

```
try {  
    ...  
}  
catch { ... }  
catch { ... } // Erreur
```

Le bloc **catch** général doit être le dernier bloc **catch** du programme, comme suit :

```
try {  
    ...  
}  
catch { ... }  
catch (OutOfMemoryException caught) { ... } // Erreur
```

L'interception d'une même classe à deux reprises génère une erreur de compilation :

```
catch (OutOfMemoryException caught) { ... }  
catch (OutOfMemoryException caught) { ... } // Erreur
```

Il en est de même si vous tentez d'intercepter une classe dérivée d'une classe interceptée dans un bloc **catch** antérieur :

```
catch (Exception caught) { ... }  
catch (OutOfMemoryException caught) { ... } // Erreur
```

Ce code provoque une erreur parce que la classe **OutOfMemoryException** est dérivée de la classe **SystemException**, elle-même dérivée de la classe **Exception**.

◆ Levée d'exceptions

- L'instruction **throw**
- La clause **finally**
- Surveillance des dépassements de capacité arithmétiques
- Indications en matière de gestion des exceptions

Le langage C# fournit l'instruction **throw** et la clause **finally** pour permettre aux programmeurs de lever des exceptions et de les gérer comme il se doit.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- lever vos propres exceptions ;
- activer la surveillance des dépassements de capacité arithmétiques.

L'instruction throw

- Lève une exception appropriée
- Associe l'exception à un message significatif

```
throw expression ;
```

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

Les blocs **try** et **catch** permettent d'intercepter les erreurs levées par un programme en C#. Vous avez vu qu'au lieu de signaler une erreur en renvoyant une valeur particulière ou en l'assignant à une variable globale d'erreur, C# transfère le contrôle d'exécution sur la clause **catch** appropriée.

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.

Attention Les programmeurs en C++ ont l'habitude de créer et de lever un objet exception à l'aide d'une instruction unique, telle que celle-ci :

```
throw out_of_range("index hors limites");
```

En C#, la syntaxe est très similaire, mais elle inclut le mot clé **new**, comme suit :

```
throw new FileNotFoundException("...");
```

Levée d'objets

Le type de l'objet levé doit être directement ou indirectement dérivé de **System.Exception**. C'est une différence par rapport au langage C++, dans lequel les objets de tout type peuvent être levés, comme dans l'exemple suivant :

```
throw 42; // Autorisé en C++, mais pas en C#
```

Vous pouvez utiliser une instruction **throw** dans un bloc **catch** pour lever à nouveau l'objet exception en cours, comme dans l'exemple suivant :

```
catch (Exception caught) {  
    ...  
    throw caught;  
}
```

Vous pouvez aussi lever un nouvel objet exception d'un autre type :

```
catch (IOException caught) {  
    ...  
    throw new FileNotFoundException(filename);  
}
```

Dans l'exemple précédent, vous remarquerez que l'objet **IOException**, et toute information qu'il contient, est perdu quand l'exception est convertie en objet **FileNotFoundException**. Il est plus judicieux d'envelopper l'exception afin d'ajouter des informations tout en conservant les informations existantes comme suit :

```
catch (IOException caught) {  
    ...  
    throw new FileNotFoundException(filename, caught);  
}
```

Cette possibilité de mapper un objet exception est particulièrement utile aux limites d'une architecture système en couches.

Vous pouvez utiliser une instruction **throw** sans expression, mais uniquement dans un bloc **catch**. Dans ce cas, l'instruction lève à nouveau l'exception en cours de traitement. Cette opération, appelée une nouvelle levée, est également possible en C++. Les deux lignes de code suivantes produisent donc un résultat identique :

```
catch (OutOfMemoryException caught) { throw caught; }  
...  
catch (OutOfMemoryException) { throw ; }
```

La clause finally

- Toutes les instructions d'un bloc finally sont toujours exécutées

```
Monitor.Enter(x);  
try {  
    ...  
}  
finally {  
    Monitor.Exit(x);  
}
```

Les blocs catch sont facultatifs

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.

Éviter la duplication d'instructions

Si les instructions qui se trouvent à la fin d'un bloc **try** sont dupliquées dans un bloc **catch** général, il suffit de les déplacer dans un bloc **finally** pour éviter la duplication. Examinez l'exemple suivant :

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


Vous pouvez réécrire ce code pour le simplifier :

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

Les instructions **break**, **continue** et **goto** ne doivent pas transférer le contrôle d'exécution à l'extérieur d'un bloc **finally**. Ces instructions ne peuvent être utilisées que si la destination se trouve dans le même bloc **finally**. En revanche, une instruction **return** ne doit jamais figurer dans un bloc **finally**, et ce même si c'est la dernière instruction du bloc.

Si l'exécution d'un bloc **finally** lève une exception, celle-ci est propagée sur le bloc **try** suivant comme suit :

```
try {  
    try {  
        ...  
    }  
    catch {  
        // ExampleException n'est pas interceptée ici  
    }  
    finally {  
        throw new ExampleException("qui m'attrapera ?");  
    }  
}  
catch {  
    // ExampleException est interceptée ici  
}
```

Si l'exécution d'un bloc **finally** lève une exception alors qu'une autre exception est en cours de propagation, l'exception d'origine est perdue :

```
try {  
    throw new ExampleException("Sera perdue");  
}  
finally {  
    throw new ExampleException("Pourrait être trouvée et interceptée");  
}
```

Surveillance des dépassements de capacité arithmétiques

■ Par défaut, le dépassement de capacité arithmétique n'est pas surveillé

- Une instruction `checked` active le contrôle du dépassement de capacité arithmétique

```
checked {  
    int number = int.MaxValue;  
    Console.WriteLine(++number);  
}
```

OverflowException

Un objet exception est levé.
WriteLine n'est pas exécutée.

```
unchecked {  
    int number = int.MaxValue;  
    Console.WriteLine(++number);  
}
```

La valeur `MaxValue + 1` est-elle négative ?

-2147483648

Par défaut, un programme C# ne contrôle pas les dépassements de capacité arithmétiques. Vous trouverez ci-dessous un exemple :

```
// example.cs  
class Example  
{  
    static void Main( )  
    {  
        int number = int.MaxValue;  
        Console.WriteLine(++number);  
    }  
}
```

Dans le code qui précède, *number* est initialisé sur la valeur maximale d'un **int**. L'expression `++number` incrémente *number* jusqu'à -2147483648, la plus grande valeur **int** négative, qui est ensuite écrite sur la console. Ce code ne génère pas de message d'erreur.

Contrôle des dépassements de capacité arithmétiques

Au moment de la compilation d'un programme C#, vous pouvez activer globalement le contrôle des dépassements de capacité arithmétiques à l'aide de l'option de ligne de commande `/checked+` comme suit :

```
c:\> csc /checked+ example.cs
```

À l'exécution, le programme générera une exception de la classe **System.OverflowException**.

De même, vous pouvez désactiver globalement le contrôle des dépassements de capacité arithmétiques à l'aide de l'option de ligne de commande `/checked-` comme suit :

```
c:\> csc /checked- example.cs
```

À l'exécution, le programme enveloppera la valeur **int** dans `int.MinValue`, et ne générera ni avertissement, ni exception de la classe **System.OverflowException**.

Création d'instructions Checked et Unchecked

Les mots clés **checked** et **unchecked** permettent de créer des instructions du même nom.

```
checked { liste-d'instructions }  
unchecked { liste-d'instructions }
```

Quel que soit le paramétrage de l'opérateur `/checked` à la compilation, le dépassement de capacité arithmétique est *toujours* contrôlé dans la liste d'instructions d'une instruction **checked**. Inversement, quel que soit le paramétrage de l'opérateur `/checked` à la compilation, le dépassement de capacité arithmétique *n'est jamais* contrôlé dans la liste d'instructions d'une instruction **unchecked**.

Création d'expressions Checked et Unchecked

Les mots clés **checked** et **unchecked** permettent également de créer des expressions du même nom :

```
checked ( expression )  
unchecked ( expression )
```

Le dépassement de capacité arithmétique est contrôlé dans une expression **checked** ; il ne l'est pas dans une expression **unchecked**. Ainsi, l'exemple suivant génère une **System.OverflowException** :

```
// example.cs  
class Example  
{  
    static void Main( )  
    {  
        int number = int.MaxValue;  
        Console.WriteLine(checked(++number));  
    }  
}
```

Indications en matière de gestion des exceptions

■ Levée

- Éviter les exceptions dans les cas normaux ou prévus
- Ne jamais créer, ni lever des objets de la classe **Exception**
- Inclure des chaînes descriptives dans les objets **Exception**
- La classe des objets levés doit être la plus spécifique possible

■ Interception

- Organisez les blocs **catch** du particulier au général
- Ne pas laisser les exceptions sortir de **Main**

Les principes suivants président à la gestion des exceptions :

- Éviter les exceptions dans les cas normaux ou prévus.
- Ne jamais créer, ni lever des objets de la classe **Exception**.
Créez uniquement des classes d'exceptions dérivées directement ou indirectement de la classe **SystemException** (jamais de la classe racine **Exception**). Vous trouverez ci-dessous un exemple :

```
class SyntaxException : SystemException
{
    ...
}
```
- Inclure des chaînes descriptives dans les objets **Exception**.
Un objet exception doit toujours inclure une chaîne descriptive significative, telle que celle-ci :

```
string description =
    String.Format("{0}({1}): saut de ligne dans une
    constante string", filename, linenumber);
throw new SyntaxException(description);
```
- La classe des objets levés doit être la plus spécifique possible.
Levez l'exception la mieux à même de fournir des informations utiles à l'utilisateur. Par exemple, levez une exception **FileNotFoundException** plutôt qu'une exception plus générale **IOException**.

- Organiser les blocs **catch** du particulier au général.

Placez les blocs **catch** en commençant par l'exception la plus spécifique et en terminant par l'exception la plus générale :

```
catch (SyntaxException caught) { ... } // Faire ceci  
catch (Exception caught) { ... }
```

- Ne pas laisser les exceptions sortir de **Main**.

Placez une clause **catch** générale dans **Main** pour vous assurer qu'aucune exception n'atteint la fin du programme sans trouver de **catch** d'interception correspondant.

```
static void Main( )  
{  
    try {  
        ...  
    }  
    catch (Exception caught) {  
        ...  
    }  
}
```

Atelier 4.2 : Utilisation des exceptions



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- lever et intercepter des exceptions ;
- afficher des messages d'erreur.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les éléments suivants :

- création de variables en C# ;
- utilisation des opérateurs courants en C# ;
- création de types **enum** en C#.

Durée approximative de cet atelier : 30 minutes

Exercice 1

Validation du numéro de jour

Au cours de cet exercice, vous ajouterez une fonctionnalité au programme que vous avez créé dans l'exercice 1. Le programme examinera le premier numéro de jour entré par l'utilisateur. Si ce nombre est inférieur à 1 ou supérieur à 365, le programme lèvera une exception **InvalidArgument** (« Jour hors limites »). Le programme interceptera ensuite cette exception dans une clause **catch** et affichera un message de diagnostic sur la console.

► Pour valider le numéro de jour

1. Ouvrez le projet WhatDay2.sln situé dans *dossier d'installation\Labs\Lab04\Starter\WhatDay2*.
2. Placez tout le contenu de **WhatDay.Main** dans un bloc **try**.
3. À la suite du bloc **try**, ajoutez une clause **catch** qui intercepte les exceptions de type **System.Exception**, et nommez-la **caught**. Dans le bloc **catch**, ajoutez une instruction **WriteLine** pour écrire sur la console l'exception interceptée.
4. Ajoutez une instruction **if** après la déclaration de la variable *dayNum*. L'instruction **if** lèvera un objet exception **new** de type **System.ArgumentOutOfRangeException** si *dayNum* est inférieur à 1 ou supérieur à 365. Utilisez le littéral **string** « Jour hors limites » pour créer l'objet exception.

5. Voici le programme complet :

```
using System;

enum MonthName { ... }

class WhatDay
{
    static void Main( )
    {
        try {
            Console.Write("Entrez un nombre compris
↪entre 1 et 365 : ");
            string line = Console.ReadLine( );
            int dayNum = int.Parse(line);

            if (dayNum < 1 || dayNum > 365) {
                throw new ArgumentOutOfRangeException("Jour
↪hors limites");
            }

            int monthNum = 0;

            foreach (int daysInMonth in DaysInMonths) {
                if (dayNum <= daysInMonth) {
                    break;
                } else {
                    dayNum -= daysInMonth;
                    monthNum++;
                }
            }
            MonthName temp = (MonthName)monthNum;
            string monthName = temp.ToString( );

            Console.WriteLine("{0} {1}", dayNum,
↪monthName);
        }
        catch (Exception caught) {
            Console.WriteLine(caught);
        }
        ...
    }
}
```

6. Enregistrez votre travail.
7. Compilez le programme WhatDay2.cs et corrigez les erreurs, le cas échéant. Exécutez le programme. Testez les données du tableau fourni dans l'Atelier4.1 (Exercice 1) pour vérifier que le programme fonctionne toujours correctement.
8. Exécutez le programme en entrant des nombres inférieurs à 1 et supérieurs à 365. Assurez-vous que les entrées incorrectes sont piégées et que l'objet exception est levé, intercepté et affiché.

Exercice 2

Gestion des années bissextiles

Au cours de cet exercice, vous ajouterez une fonctionnalité au programme que vous avez créé dans l'exercice 1. Vous inviterez l'utilisateur à entrer une année en plus d'un numéro de jour. Le programme détectera s'il s'agit d'une année bissextile. Il décidera si le numéro de jour est compris entre 1 et 366 (en cas d'année bissextile) ou entre 1 et 365 (en cas d'année non bissextile). Le programme incorporera, en dernier lieu, une nouvelle instruction **foreach** destinée à calculer la paire jour / mois pour les années bissextiles.

► Pour entrer l'année à partir de la console

1. Ouvrez le projet WhatDay3.sln situé dans *dossier d'installation\Labs\Lab04\Starter\WhatDay3*.
2. Ajoutez, au début de **WhatDay.Main**, une instruction **System.Console.Write** qui écrit sur la console une invite sollicitant l'entrée d'une année.
3. Remplacez la déclaration et l'initialisation de **string line** par une assignation. Remplacez `string line = Console.ReadLine();` par `line = Console.ReadLine();`.
4. Ajoutez à **Main** une instruction qui déclare une variable **string** nommée *line*, et initialisez-la à partir de la ligne lue sur la console avec la méthode **System.Console.ReadLine**.
5. Ajoutez à **Main** une instruction qui déclare une variable **int** nommée *yearNum*, et initialisez-la avec l'entier renvoyé par la méthode **int.Parse**.

6. Voici le programme complet :

```
using System;

enum MonthName { ... }

class WhatDay
{
    static void Main( )
    {
        try {
            Console.Write("Entrez l'année : ");
            string line = Console.ReadLine( );
            int yearNum = int.Parse(line);

            Console.Write("Entrez un nombre compris
↪ entre 1 et 365 : ");
            line = Console.ReadLine( );
            int dayNum = int.Parse(line);

            // Comme précédemment....
        }
        catch (Exception caught) {
            Console.WriteLine(caught);
        }
        ...
    }
}
```

7. Enregistrez votre travail.
8. Compilez le programme WhatDay3.cs et corrigez les erreurs, le cas échéant.

► Pour déterminer si l'année est bissextile

1. Ajoutez, tout de suite après la déclaration de *yearNum*, une instruction qui déclare une variable **bool** nommée *isLeapYear*. Initialisez cette variable avec une expression booléenne qui détermine si *yearNum* est une année bissextile. Une année est bissextile si les deux instructions suivantes sont vraies :
 - l'année est divisible par 4 ;
 - l'année *n'est pas* divisible par 100, *ou* est divisible par 400.
2. Ajoutez une instruction **if** immédiatement après la déclaration de la variable *isLeapYear*. Dans l'instruction **if**, écrivez la chaîne « EST une année bissextile » ou « N'est PAS une année bissextile », selon la valeur de *isLeapyear*. Cette instruction **if** vérifiera si la détermination booléenne de l'année bissextile est correcte.

3. Voici le programme complet :

```
using System;

enum MonthName { ... }

class WhatDay
{
    static void Main( )
    {
        try
        {
            Console.Write("Entrez l'année : ");
            string line = Console.ReadLine( );
            int yearNum = int.Parse(line);

            bool isLeapYear = (yearNum % 4 == 0)
                && (yearNum % 100 != 0
                    || yearNum % 400 == 0);

            if (isLeapYear)
            {
                Console.WriteLine(" EST une année
↳bissextile");
            } else
            {
                Console.WriteLine(" N'est PAS une année
↳bissextile");
            }

            Console.Write("Entrez un nombre compris
↳entre 1 et 365 : ");
            line = Console.ReadLine( );
            int dayNum = int.Parse(line);

            // Comme précédemment...
        }
        catch (Exception caught)
        {
            Console.WriteLine(caught);
        }
        ...
    }
}
```

4. Enregistrez votre travail.
5. Compilez le programme WhatDay3.cs et corrigez les erreurs, le cas échéant. Vérifiez si la détermination booléenne de l'année bissextile est correcte à l'aide du tableau suivant.

Année bissextile	Année non bissextile
1996	1999
2000	1900
2004	2001

6. Commentez l'instruction **if** que vous avez ajoutée à l'étape 2.

► **Pour valider le numéro de jour avec 365 ou 366**

1. Sous la déclaration de *isLeapYear*, ajoutez la déclaration d'une variable **int** nommée *maxDayNum*. Initialisez *maxDayNum* sur 366 ou 365, selon que *isLeapYear* est **true** ou **false**, respectivement.
2. Modifiez l'instruction **WriteLine** qui invite l'utilisateur à entrer un numéro de jour. Cette instruction doit indiquer une plage de 1 à 366 si une année bissextile a été entrée, et une plage de 1 à 365 dans le cas contraire.
3. Compilez le programme WhatDay3.cs et corrigez les erreurs, le cas échéant. Exécutez le programme et assurez-vous que vous avez réalisé correctement l'étape précédente.
4. Modifiez l'instruction **if** qui valide la valeur de *dayNum* de manière à utiliser la variable *maxDayNum* à la place du littéral 365.
5. Voici le programme complet :

```
using System;

enum MonthName { ... }

class WhatDay
{
    static void Main( )
    {
        try
        {
            Console.Write("Entrez l'année : ");
            string line = Console.ReadLine( );
            int yearNum = int.Parse(line);

            bool isLeapYear = (yearNum % 4 == 0)
                && (yearNum % 100 != 0
                    || yearNum % 400 == 0);

            int maxDayNum = isLeapYear ? 366 : 365;

            Console.Write("Entrez un nombre compris
↪entre 1 et {0} : ", maxDayNum);
            line = Console.ReadLine( );
            int dayNum = int.Parse(line);

            if (dayNum < 1 || dayNum > maxDayNum) {
                throw new ArgumentOutOfRangeException("Jour
↪hors limites");
            }
            // Comme précédemment...
        }
        catch (Exception caught)
        {
            Console.WriteLine(caught);
        }
    }
    ...
}
```

6. Enregistrez votre travail.
7. Compilez le programme WhatDay3.cs et corrigez les erreurs, le cas échéant. Exécutez le programme et assurez-vous que vous avez réalisé correctement l'étape précédente.

► **Pour calculer correctement la paire jour / mois pour les années bissextiles**

1. Sous l'instruction **if** qui valide le numéro de jour et la déclaration de l'entier *monthNum*, ajoutez une instruction **if-else**. Dans cette instruction, la variable *isLeapYear* servira d'expression booléenne.
2. Déplacez l'instruction **foreach** de façon à ce qu'elle soit imbriquée dans l'instruction **if-else** dans les *deux* cas : **true** et **false**. Voici le code complet :

```
if (isLeapYear)
{
    foreach (int daysInMonth in DaysInMonths) {
        ...
    }
} else
{
    foreach (int daysInMonth in DaysInMonths) {
        ...
    }
}
```

3. Enregistrez votre travail.
4. Compilez le programme WhatDay3.cs et corrigez les erreurs, le cas échéant. Exécutez le programme et vérifiez si les numéros de jour des années non bissextiles sont toujours traités correctement.
5. L'étape suivante fait appel à la collection **DaysInLeapMonths** fournie. Il s'agit d'une collection de valeurs **int** similaire à **DaysInMonths**, si ce n'est que la seconde valeur de la collection (le nombre de jours en février) est 29, et non 28.
6. Remplacez **DaysInMonth** par **DaysInLeapMonths** dans la partie **true** de l'instruction **if-else**.

7. Voici le programme complet :

```
using System;

enum MonthName { ... }

class WhatDay
{
    static void Main( )
    {
        try {
            Console.Write("Entrez l'année : ");
            string line = Console.ReadLine( );
            int yearNum = int.Parse(line);

            bool isLeapYear = (yearNum % 4 == 0)
                && (yearNum % 100 != 0
                    || yearNum % 400 == 0);

            int maxDayNum = isLeapYear ? 366 : 365;

            Console.Write("Entrez un nombre compris
↪entre 1 et {0} : ", maxDayNum);
            line = Console.ReadLine( );
            int dayNum = int.Parse(line);

            if (dayNum < 1 || dayNum > maxDayNum) {
                throw new ArgumentOutOfRangeException("Jour
↪hors limites");
            }

            int monthNum = 0;

            if (isLeapYear) {
                foreach (int daysInMonth in
↪DaysInLeapMonths) {
                    if (dayNum <= daysInMonth) {
                        break;
                    } else {
                        dayNum -= daysInMonth;
                        monthNum++;
                    }
                }
            } else {
                foreach (int daysInMonth in DaysInMonths) {
                    if (dayNum <= daysInMonth) {
                        break;
                    } else {
                        dayNum -= daysInMonth;
                        monthNum++;
                    }
                }
            }
        }
    }
}
```

Suite du code à la page suivante.

```

        MonthName temp = (MonthName)monthNum;
        string monthName = temp.ToString( );
        Console.WriteLine("{0} {1}", dayNum,
↪monthName);
    }
    catch (Exception caught) {
        Console.WriteLine(caught);
    }
}
...
}

```

8. Enregistrez votre travail.
9. Compilez le programme WhatDay3.cs et corrigez les erreurs, le cas échéant. Exécutez le programme en entrant les données du tableau suivant pour vérifier le bon fonctionnement du programme.

Année	Numéro de jour	Paire Mois / Jour
1999	32	1f février
2000	32	1 février
1999	60	1 mars
2000	60	29 février
1999	91	1 avril
2000	91	31m ars
1999	186	5 juillet
2000	186	4 juillet
1999	304	31 octobre
2000	304	30 octobre
1999	309	5 novembre
2000	309	4 novembre
1999	327	23 novembre
2000	327	22 novembre
1999	359	25 décembre
2000	359	24 décembre

Contrôle des acquis

- Introduction aux instructions
- Utilisation des instructions conditionnelles
- Utilisation des instructions d'itération
- Utilisation des instructions de saut
- Gestion des exceptions fondamentales
- Levée d'exceptions

-
1. Écrivez une instruction **if** qui évaluera si une variable **int** nommée *hour* est supérieure ou égale à zéro, et inférieure à 24. Si ce n'est pas le cas, *hour* sera réinitialisée à zéro.

 2. Écrivez une instruction **do-while** qui lit un entier sur la console, puis le stocke dans un **int** nommée *hour*. Écrivez la boucle de telle sorte que, pour en sortir, la valeur de *hour* doive être comprise entre 1 et 23 (inclus).

-
3. Écrivez une instruction **for** qui réponde à toutes les conditions de la question précédente, et qui n'autorise que cinq tentatives d'entrée d'une valeur pour *hour*. N'utilisez pas d'instruction **break** ou **continue**.
 4. Réécrivez le code correspondant à la question 3 avec une instruction **break**.
 5. Écrivez une instruction qui lève une exception de type **ArgumentOutOfRangeException** si la variable *percent* est inférieure à zéro ou supérieure à 100.

6. Le code suivant est destiné à gérer les exceptions. Expliquez pourquoi ce code est incorrect. Solutionnez le problème.

```
try
{
...
}
catch (Exception) {...}
catch (IOException) {...}
```

Module 5 : Méthodes et paramètres

Table des matières

Vue d'ensemble	1
Utilisation des méthodes	2
Utilisation des paramètres	16
Utilisation de méthodes surchargées	30
Atelier 5.1 : Création et utilisation de méthodes	38
Contrôle des acquis	50



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Utilisation des méthodes
- Utilisation des paramètres
- Utilisation de méthodes surchargées

Pour concevoir la plupart des applications, vous les divisez en unités fonctionnelles. Il s'agit là d'un principe essentiel dans la conception d'applications, car les petites sections de code sont plus faciles à comprendre, à concevoir, à développer et à déboguer. Diviser une application en unités fonctionnelles permet de réutiliser les composants fonctionnels à travers toute l'application.

En C#, vous structurez votre application en classes qui contiennent des blocs de code nommés, appelés « méthodes ». Une *méthode* est membre d'une classe qui effectue une action ou calcule une valeur.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- créer des méthodes statiques qui acceptent des paramètres et retournent des valeurs ;
- passer des paramètres à des méthodes, par différents moyens ;
- déclarer et utiliser des méthodes surchargées.

◆ Utilisation des méthodes

- Définition des méthodes
- Appel de méthodes
- Utilisation de l'instruction return
- Utilisation de variables locales
- Retour de valeurs

Dans cette section, vous apprendrez à utiliser les méthodes en C#. Les méthodes sont des mécanismes importants qui permettent de structurer le code. Vous apprendrez à créer des méthodes et à les appeler à partir d'une même classe et d'une classe vers une autre.

Vous apprendrez ensuite à utiliser des variables locales, à les allouer et à les détruire.

Enfin, vous apprendrez à retourner une valeur à partir d'une méthode et à utiliser des paramètres pour passer des données vers et à partir d'une méthode.

Définition des méthodes

- **Main est une méthode**

- Utilisez la même syntaxe pour définir vos méthodes

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

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.

Exemples de méthodes

Sur cette diapositive, 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. Elle peut être appelée à partir de votre programme. La méthode **WriteLine** est une méthode statique de la classe **System.Console**. La méthode **ExampleMethod** appartient à la classe **ExampleClass**. Cette méthode appelle la méthode **WriteLine**.

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 déclarés 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 de la diapositive.

- 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.

Syntaxe pour la définition des méthodes

Pour créer une méthode, utilisez la syntaxe suivante :

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

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");  
    }  
}
```

Remarque En C#, les noms de méthodes respectent la casse. Par conséquent, vous pouvez déclarer et utiliser des méthodes qui diffèrent seulement par la casse. Par exemple, vous pouvez déclarer des méthodes appelées **print** et des méthodes appelées **PRINT** au sein de la même classe. Toutefois, le Common Language Runtime demande que les noms de méthodes d'une classe diffèrent autrement que par la casse afin d'assurer la compatibilité avec les langages dans lesquels les noms de méthodes respectent la casse. Ce point est important à considérer si vous voulez que votre application interagisse avec des applications écrites dans des langages autres que C#.

Appel de méthodes

■ Après avoir défini une méthode, vous pouvez :

- Appeler une méthode à partir de la même classe
Utilisez le nom de la méthode suivi d'une liste de paramètres entre parenthèses
- Appeler une méthode d'une autre classe
Vous devez indiquer au compilateur la classe qui contient la méthode à appeler
La méthode appelée doit être déclarée avec le mot clé **public**
- Utiliser des appels imbriqués
Les méthodes peuvent appeler des méthodes, qui peuvent en appeler d'autres, etc.

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.

Appel de méthodes

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( );
```

Remarque à l'attention des développeurs Visual Basic Il n'existe pas d'instruction **Call**. Les parenthèses sont requises pour tous les appels de méthodes.

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");
    }
}
```

Appel de méthodes à partir d'autres classes

Pour autoriser les méthodes d'une classe à appeler des méthodes qui se trouvent dans d'autres classes, vous devez procéder comme suit :

- Spécifier la classe qui contient la méthode à appeler.
Pour spécifier la classe qui contient la méthode, utilisez la syntaxe suivante :
`ClassName.MethodName()`;
- Déclarer la méthode qui est appelée avec le mot clé **public**.

L'exemple suivant montre comment appeler la méthode **TestMethod** qui est définie dans la classe **A**, à partir de la méthode **Main** de la classe **B** :

```
using System;

class A
{
    public static void TestMethod( )
    {
        Console.WriteLine("Méthode TestMethod de la classe A");
    }
}

class B
{
    static void Main( )
    {
        A.TestMethod( );
    }
}
```

Si, dans l'exemple ci-dessus, le nom de la classe était supprimé, le compilateur rechercherait une méthode appelée **TestMethod** dans la classe **B**. Étant donné qu'il n'existe aucune méthode de ce nom dans cette classe, le compilateur afficherait le message d'erreur suivant : « Le nom 'TestMethod' n'existe pas dans la classe ni dans l'espace de noms 'B' ».

Si vous ne déclarez pas une méthode comme publique (**public**), elle devient par défaut une méthode privée (**private**) pour la classe. Par exemple, si vous omettez le mot clé **public** dans la définition de la méthode **TestMethod**, le compilateur affiche le message d'erreur suivant : « 'A.TestMethod()' est inaccessible en raison de son niveau de protection ».

Vous pouvez également utiliser le mot clé **private** pour spécifier que la méthode ne peut être appelée qu'à partir de la classe. Les deux lignes de code suivantes produisent exactement le même effet parce que les méthodes sont privées par défaut :

```
private static void MyMethod( );
static void MyMethod( );
```

Les mots clés **public** et **private** montrés ci-dessus spécifient l'*accessibilité* de la méthode. Ces mots clés contrôlent si une méthode peut être appelée depuis l'extérieur de la classe dans laquelle elle est définie.

Imbrication d'appels de méthodes

Vous pouvez aussi appeler des méthodes à partir de méthodes. L'exemple suivant montre comment imbriquer des appels de méthodes.

```
using System;
class NestExample
{
    static void Method1( )
    {
        Console.WriteLine("Method1");
    }
    static void Method2( )
    {
        Method1( );
        Console.WriteLine("Method2");
        Method1( );
    }
    static void Main( )
    {
        Method2( );
        Method1( );
    }
}
```

Le résultat de ce programme est le suivant :

```
Method1
Method2
Method1
Method1
```

Grâce à l'imbrication, il est possible d'appeler un nombre illimité de méthodes. Même s'il n'existe pas de limite prédéfinie pour le niveau d'imbrication, l'environnement d'exécution peut l'imposer, habituellement en raison de la taille de la mémoire vive (RAM) disponible nécessaire à l'exécution du processus. Pour chaque appel de méthode, la mémoire doit stocker des adresses de retour et d'autres informations.

En règle générale, si la mémoire dont vous disposez devient insuffisante pour les appels de méthodes imbriqués, vous êtes vraisemblablement confronté à un problème au niveau de la conception des classes.

Utilisation de l'instruction return

- Retour immédiat
- Utilisation de l'instruction return avec une instruction conditionnelle

```
static void ExampleMethod( )  
{  
    int numBeans;  
    //...  
  
    Console.WriteLine("Hello");  
    if (numBeans < 10)  
        return;  
    Console.WriteLine("World");  
}
```

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.

Retour immédiat

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. Si vous avez activé le niveau 2 ou supérieur pour les avertissements du compilateur C#, le compilateur affiche le message suivant : « Impossible d'atteindre le code détecté. »

Utilisation de l'instruction **return** avec une instruction conditionnelle

Il est plus courant, et beaucoup plus utile, d'utiliser l'instruction **return** dans une instruction conditionnelle telle que **if** ou **switch**. Une méthode peut ainsi retourner à l'appelant lorsqu'une condition donnée est remplie.

Dans l'exemple suivant, la méthode retourne à l'appelant si la variable *numBeans* est inférieure à 10 ; dans le cas contraire, l'exécution continue dans cette méthode.

```
static void ExampleMethod( )
{
    int numBeans;
    //...
    Console.WriteLine("Hello");
    if (numBeans < 10)
        return;
    Console.WriteLine("World");
}
```

Conseil La définition d'un point d'entrée et d'un point de sortie dans une méthode est généralement considérée comme un bon principe de programmation. La conception de C# garantit que toutes les méthodes commencent l'exécution à la première instruction. Une méthode sans instruction **return** possède un point de sortie, à la fin du bloc de code. Une méthode avec plusieurs instructions **return** possède plusieurs points de sortie, ce qui peut rendre la méthode difficile à comprendre et à modifier dans certains cas.

Retour avec une valeur

Si une méthode est définie avec un type de données plutôt qu'avec un paramètre **void**, le mécanisme de retour est utilisé pour assigner une valeur à la fonction. Ce point sera abordé plus loin dans ce module.

Utilisation de variables locales

■ Variables locales

- Créées au début de la méthode
- Privées pour la méthode
- Détruites à la sortie

■ Variables partagées

- Les variables de classe sont utilisées pour le partage

■ Conflits de portée

- Le compilateur n'envoie pas de message d'avertissement en cas de conflit entre un nom de variable locale et un nom de classe

Chaque méthode possède son propre jeu de variables locales. Vous pouvez utiliser ces variables uniquement à l'intérieur de la méthode dans laquelle elles sont déclarées. Elles ne sont pas accessibles à partir d'un autre emplacement dans l'application.

Variables locales

Vous pouvez inclure des variables locales dans le corps d'une méthode, comme le montre l'exemple suivant :

```
static void MethodWithLocals( )
{
    int x = 1; // Valeur initiale de la variable
    ulong y;
    string z;
    ...
}
```

Vous pouvez assigner aux variables locales une valeur initiale. (Par exemple, voir la variable *x* dans le code précédant.) Si vous n'assignez pas de valeur ou si vous fournissez une expression initiale pour déterminer une valeur, la variable n'est pas initialisée.

Les variables déclarées dans une méthode sont complètement séparées de celles qui sont déclarées dans d'autres méthodes, même si elles portent les mêmes noms.

La mémoire pour les variables locales est allouée à chaque fois que la méthode est appelée, puis libérée dès la fin de la méthode. Par conséquent, les valeurs stockées dans ces variables pour un appel de méthode ne sont pas conservées pour l'appel de méthode suivant.

Variables partagées

Examinez le code suivant qui tente de compter le nombre d'appels d'une méthode :

```
class CallCounter_Bad
{
    static void Init( )
    {
        int nCount = 0;
    }
    static void CountCalls( )
    {
        int nCount;
        ++nCount;
        Console.WriteLine("Méthode appelée {0} fois", nCount);
    }
    static void Main( )
    {
        Init( );
        CountCalls( );
        CountCalls( );
    }
}
```

Ce programme ne peut pas être compilé en raison de deux problèmes importants. La variable *nCount* dans **Init** n'est pas identique à la variable *nCount* dans **CountCalls**. Indépendamment du nombre de fois auquel vous appelez la méthode **CountCalls**, la valeur *nCount* est perdue chaque fois que la méthode **CountCalls** se termine.

Pour écrire ce code correctement, il faut utiliser une variable de classe, comme le montre l'exemple suivant :

```
class CallCounter_Good
{
    static int nCount;
    static void Init( )
    {
        nCount = 0;
    }
    static void CountCalls( )
    {
        ++nCount;
        Console.WriteLine("Méthode appelée " + nCount + " fois.");
    }
    static void Main( )
    {
        Init( );
        CountCalls( );
        CountCalls( );
    }
}
```

Dans cet exemple, la variable *nCount* est déclarée au niveau de la classe et non au niveau de la méthode. Par conséquent, *nCount* est partagée entre toutes les méthodes de la classe.

Conflits de portée

Dans le langage C#, vous pouvez déclarer une variable locale qui a le même nom qu'une variable de classe, mais cela peut entraîner des résultats inattendus. Dans l'exemple suivant, la variable *NumItems* est déclarée en tant que variable de la classe **ScopeDemo** et également en tant que variable locale de la méthode **Method1**. Les deux variables sont complètement différentes. Dans **Method1**, la variable *numItems* fait référence à la variable locale. Dans **Method2**, la variable *numItems* fait référence à la variable de classe.

```
class ScopeDemo
{
    static int numItems = 0;
    static void Method1( )
    {
        int numItems = 42;
        ...
    }
    static void Method2( )
    {
        numItems = 61;
    }
}
```

Conseil Étant donné que le compilateur C# n'envoie pas de message d'avertissement lorsque des variables locales et des variables de classe ont le même nom, vous pouvez utiliser une convention d'affectation de noms pour différencier les variables locales des variables de classe.

Retour de valeurs

- Déclarez la méthode avec un type de retour non void
- Ajoutez une instruction **return** avec une expression
 - Définit la valeur de retour
 - Retourne à l'appelant
- Les méthodes non void doivent retourner une valeur

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

```
int x;  
x = TwoPlusTwo( );  
Console.WriteLine(x);
```

Vous avez appris à utiliser l'instruction **return** pour terminer immédiatement une méthode. Vous allez maintenant apprendre à utiliser l'instruction **return** pour retourner une valeur à partir d'une méthode. Pour ce faire, vous devez :

1. Déclarer la méthode avec le type de valeur à retourner.
2. Ajouter une instruction **return** à l'intérieur de la méthode.
3. Inclure la valeur à retourner à l'appelant.

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;
    }

    static void Main( )
    {
        int x;
        x = TwoPlusTwo( );
        Console.WriteLine(x);
    }
}
```

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. Dans cet exemple, la valeur est stockée dans la variable locale *x* de la méthode **Main**.

Les méthodes non void doivent retourner des valeurs

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. Si le compilateur détecte qu'une méthode non void ne contient pas d'instruction **return**, il affiche le message d'erreur suivant : « Tous les chemins de code ne retournent pas nécessairement une valeur ». Ce message s'affiche également lorsque le compilateur détecte qu'il est possible d'exécuter une méthode non void sans retourner aucune valeur.

Conseil Vous pouvez uniquement utiliser l'instruction **return** pour retourner une seule valeur à partir de chaque appel de méthode. Si vous devez retourner plusieurs valeurs à partir d'un appel de méthode, vous pouvez utiliser les paramètres **ref** ou **out**. Ces paramètres sont abordés plus loin dans ce module. Une autre possibilité vous est offerte : retourner une référence à un tableau, une classe ou un struct, qui peuvent contenir plusieurs valeurs. La consigne générale qui conseille de ne pas utiliser plusieurs instructions **return** dans une même méthode s'applique également aux méthodes non void.

◆ Utilisation des paramètres

- Déclaration et appel de paramètres
- Mécanismes de passage de paramètres
- Passage par valeur
- Passage par référence
- Paramètres de sortie
- Utilisation des listes de paramètres de longueur variable
- Principes de passage de paramètres
- Utilisation de méthodes récursives

Dans cette section, vous allez apprendre à déclarer des paramètres et à appeler des méthodes avec des paramètres. Vous découvrirez également comment passer des paramètres. Enfin, vous apprendrez comment C# prend en charge les appels de méthodes récursives.

Dans cette section, vous allez apprendre à :

- Déclarer et appeler des paramètres.
- Passer des paramètres en utilisant les mécanismes suivants :
 - Passage par valeur
 - Passage par référence
 - Paramètres de sortie
- Utiliser des appels de méthodes récursives.

Déclaration et appel de paramètres

■ Déclaration des paramètres

- Placez-les entre parenthèses après le nom de la méthode
- Définissez le type et le nom de chaque paramètre

■ Appel de méthodes avec des paramètres

- Indiquez une valeur pour chaque paramètre

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

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

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. Dans les exemples présentés jusqu'à présent dans ce module, les listes de paramètres étaient vides.

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 deux façons d'appeler une méthode avec des paramètres. Dans chaque cas, 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");
```

```
int p = 7;  
string s = "Message de test";
```

```
MethodWithParameters(p, s);
```

Mécanismes de passage de paramètres

■ Trois méthodes de passage de paramètres

in	Passage par valeur
in out	Passage par référence
out	Paramètres de sortie

Il existe trois façons de passer des paramètres :

- Par valeur

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

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

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.

Passage par valeur

■ Mécanisme par défaut de passage de paramètres :

- La valeur du paramètre est copiée
- La variable peut être modifiée à l'intérieur de la méthode
- Aucune incidence sur la valeur située en dehors de la méthode
- Le paramètre doit être d'un type identique ou compatible

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

Dans les applications, la plupart des paramètres sont utilisés pour passer des informations à l'intérieur d'une méthode, et non à l'extérieur. Par conséquent, C# utilise par défaut le mécanisme de passage des paramètres 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.

L'expression fournie pour chaque paramètre par valeur doit être du même type que la déclaration du paramètre par valeur ou bien d'un type qui peut être implicitement converti dans ce type. 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** est complètement séparée de la variable *k* de la méthode **Main**. La variable *x* 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
}
```


Passage par référence

- **Définition des paramètres par référence**
 - Une référence à un emplacement dans la mémoire
- **Utilisation des paramètres par référence**
 - Utilisez le mot clé **ref** dans la déclaration et l'appel de la méthode
 - Faites correspondre les types et les valeurs des variables
 - Les modifications apportées à la méthode ont une incidence sur l'appelant
 - Assignez une valeur de paramètre avant d'appeler la méthode

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.

Déclaration des paramètres par référence

Vous pouvez déclarer un paramètre par référence en utilisant le mot clé **ref** avant le nom du type, comme le montre l'exemple suivant :

```
static void ShowReference(ref int nId, ref long nCount)
{
    // ...
}
```

Utilisation de plusieurs types de paramètres

Le mot clé **ref** s'applique uniquement au paramètre qui le suit, et non à toute la liste de paramètres. Examinons l'exemple de méthode suivant où le paramètre *nId* est passé par référence alors que le paramètre *longVar* est passé par valeur :

```
static void OneRefOneVal(ref int nId, long longVar)
{
    // ...
}
```

Correspondance des types de paramètres et des valeurs

Lorsque vous appelez une méthode, vous fournissez des paramètres par référence en utilisant le mot clé **ref** suivi d'une variable. La valeur fournie dans l'appel de la méthode doit correspondre exactement au type dans la définition de la méthode. Cette valeur doit être une variable et non une constante ou une expression calculée.

```
int x;  
long q;  
ShowReference(ref x, ref q);
```

Si vous omettez le mot clé **ref** ou si vous fournissez une constante ou une expression calculée, le compilateur rejette l'appel et un message semblable à celui-ci s'affiche : « Impossible de convertir 'int' en 'ref int' ».

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  
}
```

Cette opération fonctionne, car lorsque la méthode **AddOne** est appelée, son paramètre *x* est défini pour faire référence au même emplacement dans la mémoire que la variable *k* de la méthode **Main**. Par conséquent, l'incréméntation de *x* entraîne l'incréméntation de *k*.

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 suivant montre comment vous pouvez initialiser des paramètres par référence avant d'appeler une méthode :

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

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

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' ».

Paramètres de sortie

■ Définition des paramètres de sortie

- Les valeurs sont transférées à l'extérieur et non à l'intérieur

■ Utilisation des paramètres de sortie

- Similaires aux paramètres par référence, sauf que les valeurs ne sont pas transférées à l'intérieur de la méthode
- Utilisez le mot clé **out** dans la déclaration et l'appel de la méthode

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

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.

Utilisation des listes de paramètres de longueur variable

- Utilisez le mot clé **params**
- Déclarez-le en tant que tableau à la fin de la liste de paramètres
- Passage par valeur dans tous les cas

```
static long AddList(params long[] v)
{
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Main()
{
    long x = AddList(63,21,84);
}
```

C# offre un mécanisme pour passer des listes de paramètres de longueur variable.

Déclaration de paramètres de longueur variable

Il est parfois utile de disposer d'une méthode capable d'accepter un nombre variable de paramètres. Dans le langage C#, vous pouvez utiliser le mot clé **params** pour spécifier une liste de paramètres de longueur variable. Lorsque vous déclarez un paramètre de longueur variable, vous devez :

- déclarer un seul paramètre **params** par méthode ;
- placer le paramètre à la fin de la liste de paramètres ;
- déclarer le paramètre en tant que type tableau à une seule dimension.

L'exemple suivant montre comment déclarer une liste de paramètres à longueur variable.

```
static long AddList(params long[] v)
{
    long total;
    long i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
```

Étant donné qu'un paramètre **params** est toujours un tableau, toutes les valeurs doivent être du même type.

Passage de valeurs

Lorsque vous appelez une méthode avec un paramètre de longueur variable, vous pouvez passer des valeurs au paramètre **params** sous l'une des deux formes suivantes :

- Sous forme de liste d'éléments séparés par des virgules (la liste peut être vide)
- Sous forme de tableau

Le code suivant montre ces deux techniques. Elles sont toutes deux traitées de la même façon par le compilateur.

```
static void Main( )
{
    long x;
    x = AddList(63, 21, 84); // Liste
    x = AddList(new long[] { 63, 21, 84 }); // Tableau
}
```

Quelle que soit la méthode utilisée pour appeler la méthode, le paramètre **params** est traité comme un tableau. Vous pouvez utiliser la propriété **Length** du tableau pour déterminer le nombre de paramètres passés à chaque appel.

Dans un paramètre **params**, une copie des données est effectuée, et bien qu'il soit possible de modifier les valeurs à l'intérieur de la méthode, les valeurs à l'extérieur de la méthode ne sont pas modifiées.

Principes de passage de paramètres

■ Mécanismes

- Le passage par valeur est le plus courant
- La valeur de retour de la méthode est utile pour les valeurs uniques
- Utilisez le passage par référence et/ou de sortie pour retourner plusieurs valeurs
- Utilisez uniquement le passage par référence pour transférer les données dans les deux sens

■ Efficacité

- Le passage par valeur est généralement le plus efficace

Il n'est pas évident de faire le bon choix parmi les nombreuses options qui permettent de passer des paramètres. Lorsque vous décidez de passer des paramètres, tenez compte des deux facteurs suivants pour choisir la meilleure méthode : mécanismes et efficacité.

Mécanismes

Les paramètres par valeur offrent une protection limitée contre les modifications involontaires des valeurs des paramètres, car les modifications effectuées à l'intérieur d'une méthode n'ont aucun effet à l'extérieur. Il est donc conseillé d'utiliser des paramètres par valeur, sauf si vous devez impérativement passer des données à l'extérieur d'une méthode.

Si vous devez passer des données à l'extérieur d'une méthode, vous pouvez utiliser l'instruction **return**, des paramètres par référence ou des paramètres de sortie. L'instruction **return** est simple à utiliser, mais elle ne permet de retourner qu'un seul résultat. Si vous devez retourner plusieurs valeurs, vous devez utiliser les types de paramètres par référence et de sortie. Utilisez le mot clé **ref** pour transférer des données dans les deux sens et le mot clé **out** pour transférer des données à l'extérieur de la méthode.

Efficacité

D'une façon générale, les types simples comme **int** et **long** sont passés efficacement par valeur.

L'efficacité n'est pas prise en compte par le langage et vous ne devez donc pas tout baser sur elle. S'il s'agit d'une considération importante dont il faut tenir compte dans les grandes applications qui exigent de nombreuses ressources, il est généralement préférable de s'attacher à l'exactitude, à la stabilité et à la robustesse du programme. Le respect des bonnes pratiques de programmation est plus important que l'efficacité.

Utilisation de méthodes récursives

- Une méthode peut s'appeler elle-même
 - Directement
 - Indirectement
- Utile pour résoudre certains problèmes

Une méthode peut s'appeler elle-même. Cette technique est appelée la *récursivité*. Elle permet de résoudre plusieurs types de problèmes. Les méthodes récursives sont souvent utiles pour manipuler des structures de données plus complexes telles que les listes et les arborescences.

En C#, les méthodes peuvent être mutuellement récursives. Par exemple, une méthode A peut appeler une méthode B, et inversement.

Exemple de méthode récursive

La suite de Fibonacci se retrouve souvent dans le domaine des mathématiques et de la biologie (par exemple, pour calculer le taux de reproduction et la population des lapins). Le n ième membre de cette suite a la valeur 1 si n est égal à 1 ou 2 ; sinon, il est égal à la somme des deux nombres précédents dans la suite. Notez que lorsque n est supérieur à deux, la valeur du n ième membre de la suite est dérivée des valeurs de deux valeurs précédentes de la suite. Lorsque la définition d'une méthode fait référence à la méthode elle-même, la récursivité peut intervenir.

Vous pouvez implémenter la méthode **Fibonacci** comme suit :

```
static ulong Fibonacci(ulong n)
{
    if (n <= 2)
        return 1;
    else
        return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Notez que deux appels sont effectués vers la méthode à partir de la méthode elle-même.

Une méthode récursive doit contenir une condition de fin qui permet le retour de valeurs sans appels supplémentaires. Dans le cas de la méthode **Fibonacci**, le test $n \leq 2$ est la condition de fin.

◆ Utilisation de méthodes surchargées

- Déclaration de méthodes surchargées
- Signatures des méthodes
- Utilisation de méthodes surchargées

Les méthodes d'une classe ne peuvent pas avoir le même nom que d'autres éléments qui ne sont pas des méthodes dans cette classe. En revanche, plusieurs méthodes d'une classe peuvent partager le même nom. Le partage d'un nom par plusieurs méthodes est appelé la surcharge.

Dans cette section, vous allez apprendre :

- à déclarer des méthodes surchargées ;
- comment C# utilise des signatures pour différencier les méthodes qui portent le même nom ;
- à quel moment utiliser des méthodes surchargées.

Déclaration de méthodes surchargées

■ Méthodes partageant le même nom au sein d'une classe

- L'examen des listes de paramètres permet de les distinguer

```
class OverloadingExample
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(Add(1,2) + Add(1,2,3));
    }
}
```

Les méthodes surchargées sont les méthodes d'une même classe qui portent le même nom. Le compilateur C# distingue les méthodes surchargées en comparant les listes de paramètres.

Exemples de méthodes surchargées

Le code qui suit montre comment vous pouvez utiliser plusieurs méthodes portant le même nom dans une même classe :

```
class OverloadingExample
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(Add(1,2) + Add(1,2,3));
    }
}
```

Le compilateur C# recherche deux méthodes appelées **Add** dans la classe et deux appels de méthode aux méthodes appelées **Add** dans **Main**. Bien que les noms des méthodes soient identiques, le compilateur peut faire la différence entre les deux méthodes **Add** en comparant les listes de paramètres.

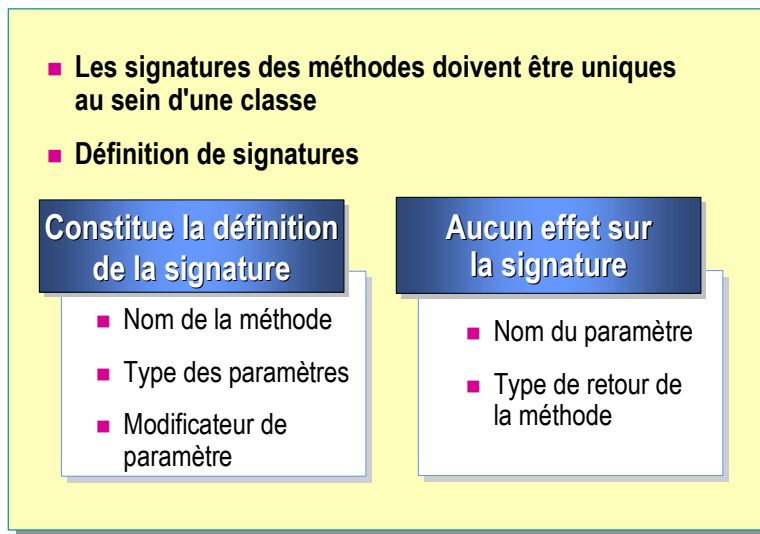
La première méthode **Add** accepte deux paramètres, tous deux du type **int**. La seconde méthode **Add** en accepte trois, également du type **int**. Étant donné que les listes de paramètres sont différentes, le compilateur accepte que les deux méthodes soient définies dans la même classe.

La première instruction dans **Main** incluant un appel à la méthode **Add** avec deux paramètres **int**, le compilateur traduit cette syntaxe comme un appel à la première méthode **Add**. Le second appel à la méthode **Add** incluant trois paramètres **int**, le compilateur l'interprète comme un appel à la seconde méthode **Add**.

Vous ne pouvez pas partager un nom entre des méthodes et des variables, des constantes ou des types énumérés dans une même classe. Il sera impossible de compiler le code ci-dessous, car le nom *k* est utilisé à la fois pour une méthode et pour une variable de classe :

```
class BadMethodNames
{
    static int k;
    static void k( ) {
        // ...
    }
}
```

Signatures des méthodes



Le compilateur C# utilise des signatures pour différencier les méthodes d'une classe. Dans chaque classe, la signature de chaque méthode doit différer de celles des autres méthodes déclarées dans cette classe.

Définition de signatures

La signature d'une méthode est composée du nom de la méthode, du nombre de paramètres qu'accepte la méthode, ainsi que du type et du modificateur (**out** ou **ref**) de chaque paramètre.

Les trois méthodes suivantes ont des signatures différentes et peuvent donc être déclarées dans la même classe.

```
static int LastErrorCode( )
{
}

static int LastErrorCode(int n)
{
}

static int LastErrorCode(int n, int p)
{
}
```

Éléments qui n'affectent pas la signature

La signature d'une méthode *n'inclut pas* le type de retour. Les deux méthodes suivantes ont la même signature et ne peuvent donc pas être déclarées dans la même classe.

```
static int LastErrorCode(int n)
{
}
static string LastErrorCode(int n)
{
}
```

La signature de la méthode *n'inclut pas* les noms des paramètres. Les deux méthodes suivantes ont la même signature, même si les noms de leurs paramètres sont différents.

```
static int LastErrorCode(int n)
{
}
static int LastErrorCode(int x)
{
}
```

Utilisation de méthodes surchargées

- **Utilisez les méthodes surchargées pour :**
 - utiliser des méthodes similaires qui nécessitant des paramètres différents ;
 - ajouter une nouvelle fonctionnalité à un code existant.
- **N'en abusez pas, car elles sont :**
 - difficiles à déboguer ;
 - difficiles à gérer.

Les méthodes surchargées sont utiles lorsque vous disposez de deux méthodes similaires qui nécessitent un nombre ou des types de paramètres différents.

Méthodes similaires nécessitant des paramètres différents

Supposez une classe contenant une méthode qui envoie un message de bienvenue à un utilisateur. Le nom de l'utilisateur est parfois connu, parfois non. Vous pouvez définir deux méthodes différentes appelées **Greet** et **GreetUser**, comme le montre le code suivant :

```
class GreetDemo
{
    static void Greet( )
    {
        Console.WriteLine("Bonjour");
    }
    static void GreetUser(string Name)
    {
        Console.WriteLine("Bonjour " + Nom);
    }
    static void Main( )
    {
        Greet( );
        GreetUser("Alex");
    }
}
```

Ce code fonctionnera, mais la classe possède désormais deux méthodes qui effectuent presque exactement la même tâche, mais sous des noms différents. Vous pouvez réécrire cette classe en surchargeant les méthodes comme le montre le code suivant :

```
class GreetDemo
{
    static void Greet( )
    {
        Console.WriteLine("Bonjour");
    }
    static void Greet(string Name)
    {
        Console.WriteLine("Bonjour " + Nom);
    }
    static void Main( )
    {
        Greet( );
        Greet("Alex");
    }
}
```

Ajout d'une nouvelle fonctionnalité à un code existant

La surcharge de méthodes est utile lorsque vous souhaitez ajouter de nouvelles fonctionnalités à une application existante sans trop modifier le code existant. Par exemple, le code précédent peut être développé en ajoutant une autre méthode dont l'action consiste à accueillir un utilisateur avec un message de bienvenue qui change en fonction de l'heure du jour, comme le montre le code suivant :

```
class GreetDemo
{
    enum TimeOfDay { Morning, Afternoon, Evening }

    static void Greet( )
    {
        Console.WriteLine("Bonjour");
    }
    static void Greet(string Name)
    {
        Console.WriteLine("Bonjour " + Name);
    }
    static void Greet(string Name, TimeOfDay td)
    {
        string Message = "";

        switch(td)
        {
```

(suite du code à la page suivante)


```
        case TimeOfDay.Morning:
            Message="Bonne journée";
            break;
        case TimeOfDay.Afternoon:
            Message="Bon après-midi";
            break;
        case TimeOfDay.Evening:
            Message="Bonsoir";
            break;
    }
    Console.WriteLine(Message + " " + Nom);
}
static void Main( )
{
    Greet( );
    Greet("Alex");
    Greet("Sandra", TimeOfDay.Morning);
}
}
```

À quel moment utiliser la surcharge

La surcharge abusive des méthodes peut rendre les classes difficiles à gérer et à corriger. D'une façon générale, il est conseillé de surcharger uniquement les méthodes dont les fonctions sont très proches mais qui diffèrent par le nombre ou le type de données dont elles ont besoin.

Atelier 5.1 : Création et utilisation de méthodes



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer et appeler des méthodes avec et sans paramètres ;
- utiliser différents mécanismes pour passer des paramètres.

Conditions préalables

Pour pouvoir réaliser cet atelier, vous devez être familiarisé avec les concepts suivants :

- la création et l'utilisation de variables ;
- les instructions C#.

Durée approximative de cet atelier : 60 minutes

Exercice 1

Utilisation de paramètres dans des méthodes qui retournent des valeurs

Dans cet exercice, vous allez définir et utiliser des paramètres d'entrée dans une méthode qui retourne une valeur. Vous écrirez également une infrastructure de test pour lire deux valeurs à partir de la console et afficher les résultats.

Vous créerez une classe appelée **Utils**. Dans cette classe, vous créerez une méthode appelée **Greater**. Cette méthode acceptera deux paramètres entiers en entrée et retournera la valeur du paramètre le plus élevé.

Pour tester la classe, vous créerez une autre classe appelée **Test**, qui demande à l'utilisateur de saisir deux nombres, appelle la méthode **Utils.Greater** pour déterminer quel est le plus grand des deux nombres, puis affiche le résultat.

► Pour créer la méthode **Greater**

1. Ouvrez le projet **Utils.sln** dans le dossier *dossier_installation\Labs\Lab05\Starter\Utility*.

Il contient un espace de noms appelé **Utils**, qui contient lui-même une classe également appelée **Utils**. Vous enregistrerez la méthode **Greater** dans cette classe.

2. Pour créer la méthode **Greater**, procédez comme suit :
 - a. Ouvrez la classe **Utils**.
 - b. Ajoutez dans la classe **Utils** une méthode publique statique appelée **Greater**.
 - c. Cette méthode acceptera deux paramètres **int**, appelés *a* et *b*, qui seront passés par valeur. La méthode retournera une valeur **int** qui représente le plus grand des deux nombres.

Le code de la classe **Utils** doit ressembler à ce qui suit :

```
namespace Utils
{
    using System;

    class Utils
    {
        //
        // Renvoie la plus élevée de deux valeurs entières
        //

        public static int Greater(int a, int b)
        {
            if (a > b)
                return a;
            else
                return b;
        }
    }
}
```

► **Pour tester la méthode Greater**

1. Ouvrez la classe **Test**.
2. Dans la méthode **Main**, écrivez le code suivant.
 - a. Définissez deux variables de type entier appelées *x* et *y*.
 - b. Ajoutez des instructions qui lisent deux entiers à partir de saisies clavier et utilisez-les pour les assigner à *x* et *y*. Utilisez les méthodes **Console.ReadLine** et **int.Parse** présentées dans les modules précédents.
 - c. Définissez un autre entier appelé *greater*.
 - d. Testez la méthode **Greater** en l'appelant et assignez la valeur de retour à la variable *greater*.

3. Écrivez le code qui permet d'afficher le plus élevé des deux entiers en utilisant la méthode **Console.WriteLine**.

Le code de la classe **Test** doit ressembler à ce qui suit :

```
namespace Utils
{
    using System;

    /// <summary>
    ///   Série de tests
    /// </summary>

    public class Test
    {
        public static void Main()
        {
            int x;          // Valeur d'entrée 1
            int y;          // Valeur d'entrée 2
            int greater;    // Résultat de Greater( )

            // Lit les nombres en entrée
            Console.WriteLine("Entrer le premier
↪nombre : ");
            x = int.Parse(Console.ReadLine( ));
            Console.WriteLine("Entrer le second
↪nombre: ");
            y = int.Parse(Console.ReadLine( ));

            // Teste la méthode Greater( )
            greater = Utils.Greater(x,y);
            Console.WriteLine("La valeur la plus élevée
↪est "+ greater);

        }
    }
}
```

4. Enregistrez votre travail
5. Compilez le projet et corrigez les erreurs éventuelles. Exécutez et testez le programme.

Exercice 2

Utilisation des méthodes avec des paramètres par référence

Dans cet exercice, vous allez écrire une méthode appelée **Swap** qui permettra d'échanger les valeurs de ses paramètres. Vous utiliserez des paramètres passés par référence.

► **Pour créer la méthode Swap**

1. Ouvrez le projet `Utils.sln` dans le dossier `dossier_installation\Labs\Lab05\Starter\Utility`, si cela n'est pas déjà fait.
2. Ajoutez la méthode **Swap** à la classe **Utils** en procédant comme suit :
 - a. Ajoutez une méthode publique statique void appelée **Swap**.
 - b. Cette méthode acceptera deux paramètres **int**, appelés *a* et *b*, lesquels seront passés par référence.
 - c. Écrivez à l'intérieur du corps de la méthode **Swap** des instructions qui échangent les valeurs *a* et *b*. Vous devrez créer une variable locale **int** dans la méthode **Swap** pour conserver temporairement l'une des deux valeurs pendant l'échange. Nommez cette variable *temp*.

Le code de la classe **Utils** doit ressembler à ce qui suit :

```
namespace Utils
{
    using System;

    public class Utils
    {
        ... code existant omis pour plus de clarté ...

        //
        // Échange deux entiers, passés par référence
        //

        public static void Swap(ref int a, ref int b)
        {
            int temp = a;
            a = b;
            b = temp;
        }
    }
}
```

► Pour tester la méthode Swap

1. Modifiez la méthode **Main** dans la classe **Test** en suivant les étapes ci-dessous :

- a. Assignez une valeur aux variables *x* et *y* de type entier.
- b. Appelez la méthode **Swap**, en passant ces valeurs en tant que paramètres.

Affichez les nouvelles valeurs des deux entiers avant et après les avoir échangés. Le code de la classe **Test** doit ressembler à ce qui suit :

```
namespace Utils
{
    using System;

    public class Test
    {
        public static void Main()
        {
            ... code existant omis pour plus de clarté ...

            // Teste la méthode Swap
            Console.WriteLine("Avant l'échange: " + x +
↵", " + y);
            Utils.Swap(ref x, ref y);
            Console.WriteLine("Après l'échange: " + x +
↵", " + y);

        }
    }
}
```

2. Enregistrez votre travail
3. Compilez le projet et corrigez les erreurs éventuelles. Exécutez et testez le programme.

Conseil Si les paramètres n'ont pas été échangés comme attendu, assurez-vous que vous les avez passés en tant que paramètres **ref**.

Exercice 3

Utilisation de méthodes avec des paramètres de sortie

Dans cet exercice, vous allez définir et utiliser une méthode statique avec un paramètre de sortie.

Vous allez également écrire une nouvelle méthode appelée **Factorial** qui accepte une valeur **int** et calcule sa factorielle. La factorielle d'un nombre est le produit de tous les nombres compris entre 1 et ce nombre. La factorielle de zéro est 1. Vous trouverez ci-après des exemples de factorielles :

- $\text{Factorielle}(0) = 1$
- $\text{Factorielle}(1) = 1$
- $\text{Factorielle}(2) = 1 * 2 = 2$
- $\text{Factorielle}(3) = 1 * 2 * 3 = 6$
- $\text{Factorielle}(4) = 1 * 2 * 3 * 4 = 24$

► Pour créer la méthode **Factorial**

1. Ouvrez le projet `Utils.sln` dans le dossier `dossier_installation\Labs\Lab05\Starter\Utility`, si cela n'est pas déjà fait.
2. Ajoutez la méthode **Factorial** à la classe **Utils** comme suit :
 - a. Ajoutez une nouvelle méthode publique statique appelée **Factorial**.
 - b. Cette méthode acceptera deux paramètres appelés n et $answer$. Le premier, passé par valeur, est une valeur **int** pour laquelle la factorielle doit être calculée. Le second paramètre est un paramètre **out int** qui sera utilisé pour retourner le résultat.
 - c. La méthode **Factorial** doit retourner une valeur **bool** qui indique si la méthode a réussi ou non. (Elle peut générer une erreur de dépassement et déclencher une exception.)
3. Ajoutez une fonctionnalité à la méthode **Factorial**.

Le moyen le plus simple pour calculer une factorielle consiste à utiliser une boucle. Pour ajouter une fonctionnalité à la méthode, suivez les étapes ci-après :

- a. Créez une variable **int** appelée k dans la méthode **Factorial**. Elle servira de compteur de boucles.
- b. Créez une autre variable **int** appelée f , qui sera utilisée comme valeur de travail à l'intérieur de la boucle. Initialisez la variable de travail f avec la valeur 1.
- c. Utilisez une boucle **for** pour effectuer l'itération. Commencez par la valeur 2 pour k et terminez lorsque k atteint la valeur du paramètre n . Incrémentez k chaque fois que la boucle est effectuée.
- d. Dans le corps de la boucle, multipliez f successivement par chaque valeur de k , en stockant le résultat dans f .

- e. Les résultats des factorielles peuvent être volumineux même lorsque les valeurs d'entrée sont petites. Vous devez donc vous assurer que tous les calculs d'entiers se trouvent dans des blocs vérifiés et que vous avez intercepté les exceptions, telles que les erreurs de dépassement arithmétiques.
- f. Assignez la valeur du résultat dans *f* au paramètre de sortie *answer*.
- g. La méthode retourne la valeur **true** si le calcul réussit, et **false** dans le cas contraire, c'est-à-dire si une exception se produit.

Le code de la classe **Utils** doit ressembler à ce qui suit :

```
namespace Utils
{
    using System;

    public class Utils
    {
        ... code existant omis pour plus de clarté ...

        //
        // Calcule la factorielle
        // et retourne le résultat sous la forme d'un paramètre
        ↪ de sortie
        //

        public static bool Factorial(int n, out int answer)
        {
            int k;          // Compteur de boucles
            int f;          // Valeur de travail
            bool ok=true;    // True en cas de réussite, sinon
            ↪ false

            // Vérifie la valeur d'entrée

            if (n<0)
                ok = false;
        }
    }
}
```

(suite du code à la page suivante)

```

        // Calcule la valeur de la factorielle comme étant
        // le produit de tous les nombres compris entre 2
        // et n

        try
        {
            checked
            {
                f = 1;
                for (k=2; k<=n; ++k)
                {
                    f = f * k;
                }
            }
        }
        catch(Exception)
        {
            // En cas d'erreur dans le calcul,
            // on l'intercepte ici. Toutes les exceptions
            // sont traitées de la même façon : on affecte
            // au résultat la valeur zéro et on retourne
            // false.

            f = 0;
            ok = false;
        }

        // Assigne la valeur de retour
        answer = f;
        // Retour à l'appelant
        return ok;
    }

}

```

► Pour tester la méthode Factorial

1. Éditez la classe **Test** comme suit :
 - a. Déclarez une variable **bool** appelée *ok* pour contenir le résultat **true** ou **false**.
 - b. Déclarez une variable **int** appelée *f* pour contenir le résultat de la factorielle.
 - c. Demandez à l'utilisateur de saisir un entier. Assignez la valeur de l'entrée à la variable *x* **int**.
 - d. Appelez la méthode **Factorial** en passant *x* comme premier paramètre et *f* comme second paramètre. Retournez le résultat dans *ok*.
 - e. Si *ok* a la valeur **true**, affichez les valeurs de *x* et de *f*; si cela n'est pas le cas, affichez un message signalant qu'une erreur s'est produite.

Le code de la classe **Test** doit ressembler à ce qui suit :

```
namespace Utils
{
    public class Test
    {
        static void Main( )
        {
            int f;          // Résultat de la factorielle
            bool ok;        // Réussite ou échec de la factorielle

            ... code existant omis pour plus de clarté ...

            // Lit l'entrée de Factorial

            Console.WriteLine("Nombre à mettre en
↪factorielle : ");
            x = int.Parse(Console.ReadLine( ));

            // Teste la fonction factorielle
            ok = Utils.Factorial(x, out f);
            // Affiche le résultat de la factorielle
            if (ok)
                Console.WriteLine("Factorielle(" + x + ") =
↪" + f);
            else
                Console.WriteLine("Impossible de calculer
↪cette factorielle");
        }
    }
}
```

2. Enregistrez votre travail
3. Compilez le programme, corrigez les erreurs éventuelles, puis exécutez et testez le programme.

S'il vous reste du temps

Implémentation d'une méthode en utilisant la récursivité

Dans cet exercice, vous allez réimplémenter la méthode **Factorial** que vous avez créée à l'exercice 3 en utilisant la récursivité plutôt que l'itération.

La factorielle d'un nombre peut être définie de façon récursive comme suit : la factorielle de zéro est 1 et vous pouvez trouver la factorielle de tout entier long en multipliant cet entier par la factorielle du nombre qui le précède. En résumé :

Si $n=0$, alors $\text{Factorielle}(n) = 1$; sinon, $n * \text{Factorielle}(n-1)$

► Pour modifier la méthode **Factorial** existante

1. Éditez la classe **Utils** et modifiez la méthode **Factorial** existante de façon à ce qu'elle utilise la récursivité plutôt que l'itération.

Les paramètres et les types des valeurs retournées seront identiques, mais la fonction interne de la méthode sera différente. Si vous souhaitez conserver la solution de l'exercice 3, vous devez utiliser un autre nom pour cette méthode.

2. Utilisez le pseudo-code indiqué plus haut pour implémenter le corps de la méthode **Factorial**. (Vous devrez le convertir en syntaxe C#.)
3. Ajoutez du code dans la classe **Test** pour tester votre nouvelle méthode.
4. Enregistrez votre travail.

5. Compilez le programme, corrigez les erreurs éventuelles, puis exécutez et testez le programme.

La version récursive de la méthode **Factorial (RecursiveFactorial)** est présentée ci-dessous :

```
//
// Autre façon de résoudre le problème de la
// factorielle, il s'agit cette fois d'une fonction
// récursive
//

public static bool RecursiveFactorial(int n, out int f)
{
    bool ok=true;

    // Intercepte les entrées négatives
    if (n<0)
    {
        f=0;
        ok = false;
    }

    if (n<=1)
        f=1;
    else
    {
        try
        {
            int pf;
            checked
            {
                ok = RecursiveFactorial(n-1,out pf);
                f = n * pf;
            }
        }
        catch(Exception)
        {
            // Une erreur s'est produite. On définit un
            // flag d'erreur et on retourne zéro.
            f=0;
            ok=false;
        }
    }

    return ok;
}
```

Contrôle des acquis

- Utilisation des méthodes
- Utilisation des paramètres
- Utilisation de méthodes surchargées

-
1. Expliquez ce que sont les méthodes et quelle est leur importance.
 2. Indiquez les trois façons de passer des données dans des paramètres, ainsi que les mots clés C# correspondants.
 3. À quel moment les variables locales sont-elles créées et détruites ?

-
4. Quel mot clé doit être ajouté à la définition d'une méthode qui doit être appelée à partir d'une autre classe ?
 5. Quels éléments d'une méthode sont utilisés pour former une signature ?
 6. Définissez la signature d'une méthode statique appelée **Rotate** qui ne retourne pas de valeur, mais qui doit « effectuer une rotation vers la droite » de ses trois paramètres de type entier.

Module 6 : Tableaux

Table des matières

Vue d'ensemble	1
Vue d'ensemble des tableaux	2
Création de tableaux	10
Utilisation des tableaux	17
Atelier 6.1 : Création et utilisation de tableaux	29
Contrôle des acquis	41



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Vue d'ensemble des tableaux
- Création de tableaux
- Utilisation des tableaux

Les tableaux constituent un moyen important pour grouper des données. Pour tirer le meilleur parti de C#, il est important de savoir comment utiliser et créer des tableaux efficacement.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- créer, initialiser et utiliser des tableaux de rangs différents ;
- utiliser des arguments de ligne de commande dans un programme en C# ;
- décrire la relation entre une variable de tableau et une instance de tableau ;
- utiliser des tableaux comme paramètres pour les méthodes ;
- renvoyer des tableaux à partir de méthodes.

◆ Vue d'ensemble des tableaux

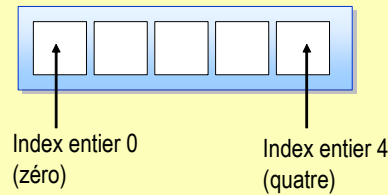
- Qu'est ce qu'un tableau ?
- Notation de tableau en C#
- Rang de tableau
- Accès aux éléments d'un tableau
- Vérification des limites de tableau
- Comparaison entre tableaux et collections

Cette section offre une vue d'ensemble des concepts généraux des tableaux, présente la syntaxe employée pour déclarer ces derniers en C# et décrit des fonctionnalités de base, telles que le rang et les éléments. Dans la section suivante, vous apprendrez à définir et à utiliser des tableaux.

Qu'est ce qu'un tableau ?

■ Un tableau est une suite d'éléments

- Tous les éléments d'un tableau sont du même type
- Les structs peuvent comporter des éléments de types différents
- L'accès aux éléments individuels se fait à l'aide d'index entiers



Il existe deux moyens fondamentaux de grouper des données connexes : les structures (**structs**) et les tableaux.

- Les structures sont des groupes de données connexes de type différent.

Par exemple, un nom (**string**), âge (**int**) et sexe (**enum**) se regroupent naturellement dans une **structure** décrivant une personne. Vous pouvez accéder aux membres individuels d'une structure à l'aide du nom de leurs champs.

- Les tableaux sont des séquences de données de même type.

Par exemple, une suite, ou alignement, de maisons se regroupent naturellement pour former une rue. 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.

Notation de tableau en C#

■ Vous déclarez une variable tableau en spécifiant :

- le type des éléments du tableau ;
- le rang du tableau ;
- le nom de la variable.

`type[] name;`

Spécifie le nom de la variable tableau

Spécifie le rang du tableau

Spécifie le type d'élément du tableau

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. D'autres langages, comme Microsoft® Visual Basic®, utilisent des parenthèses.

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

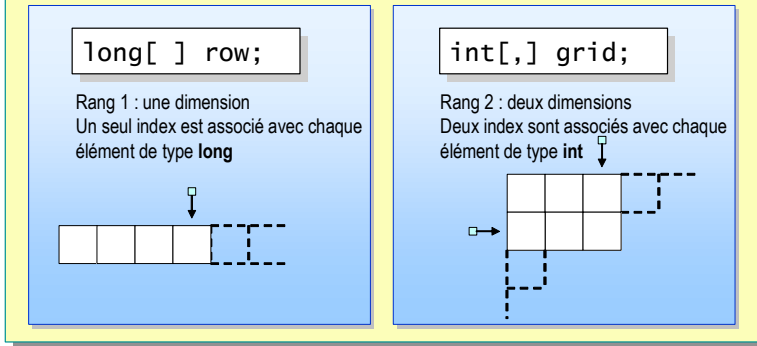
- Vous ne pouvez pas écrire de crochets à droite du nom de la variable.
- Vous ne spécifiez pas la taille du tableau lorsque vous déclarez une variable tableau.

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#
```

Rang de tableau

- Le rang est aussi qualifié de dimension du tableau
- Le nombre d'index associés avec chaque élément



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

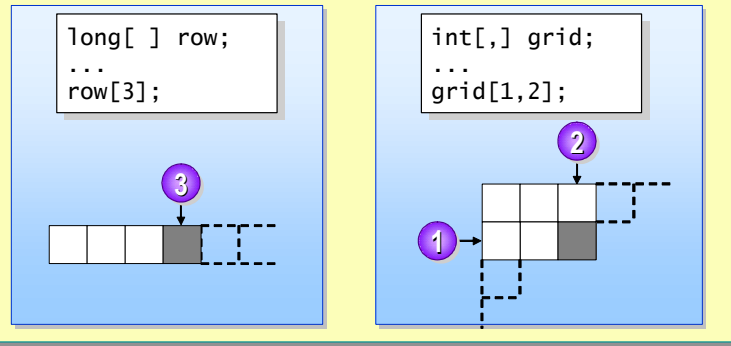
Pour déclarer un tableau à deux dimensions, vous utilisez une virgule à l'intérieur des crochets, comme illustré sur la diapositive. Un tableau de ce type s'appelle également tableau de rang 2, car deux index entiers sont associés à chaque élément du tableau. Cette notation s'étend simplement comme suit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

Accès aux éléments d'un tableau

■ Indiquez un index entier pour chaque rang

- Les index commencent à zéro



Pour accéder aux éléments d'un tableau, vous utilisez une syntaxe semblable à celle que vous utilisez pour déclarer les variables tableau. Toutes deux utilisent des crochets. Cette ressemblance (voulue, dans la mouvance des langages C et C++) peut être source de confusion, si vous n'y êtes pas familiarisé. Par conséquent, il est important que vous soyez capable de faire la distinction entre une déclaration de variable tableau et une expression d'accès à un élément de 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. Cette notation s'étend de la même manière que la notation employée pour déclarer les variables. Pour accéder à un élément à l'intérieur d'un tableau de rang n , utilisez n index entiers séparés par des virgules. Notez à nouveau que la syntaxe utilisée dans une expression d'accès à un élément de tableau reflète la syntaxe utilisée pour déclarer les variables.

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]
```

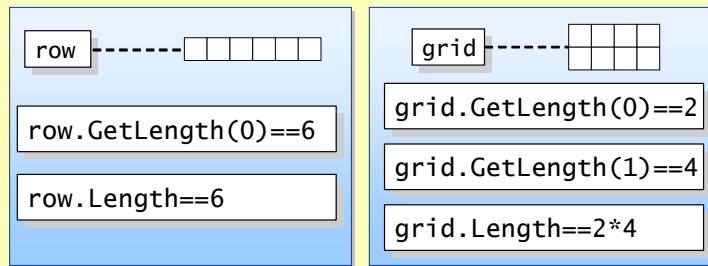

Certains programmeurs utilisent l'expression « élément initial » plutôt que « premier élément » pour réduire les risques de confusion. Indexer à partir de 0 signifie que le dernier élément d'une instance de tableau contenant *taille* éléments se trouve à [*taille-1*] et non à [*taille*]. L'utilisation accidentelle de [*taille*] est une erreur commune de décalage d'une unité, particulièrement de la part de programmeurs habitués à un langage qui indexe à partir de un, comme Visual Basic.

Remarque Bien que la technique soit rarement employée, il est possible de créer des tableaux qui possèdent des index entiers définis par l'utilisateur pour leurs limites inférieures. Pour plus d'informations, recherchez « `Array.CreateInstance` » dans l'aide du SDK Microsoft .NET Framework.

Vérification des limites de tableau

- Les limites sont contrôlées à chaque tentative d'accès à un tableau

- Un index erroné renvoie une exception `IndexOutOfRangeException`
- Utilisez la propriété **Length** et la méthode **GetLength**



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#.

Même si les limites de tableau sont automatiquement vérifiées, vérifiez tout de même que les index entiers sont toujours à l'intérieur des limites. Pour ce faire, 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]);  
    }  
}
```

Comparaison entre tableaux et collections

- **Un tableau ne peut pas se redimensionner lorsqu'il est rempli**
 - Une classe de collection, telle que `ArrayList`, peut être redimensionnée
- **Un tableau doit contenir des éléments d'un seul type**
 - Une collection est conçue pour stocker des éléments de types différents
- **L'accès aux éléments d'un tableau ne peut pas être en lecture seule**
 - L'accès à une collection peut être en lecture seule
- **En règle générale, les tableaux sont plus rapides mais moins souples**
 - Les collections sont un peu plus lentes, mais aussi plus souples

La taille d'une instance de tableau et le type des éléments qu'elle contient sont définitivement définis lors de la création du tableau. Pour créer un tableau qui contient toujours 42 éléments de type **int** (entier), utilisez la syntaxe suivante :

```
int[ ] rigid = new int [ 42 ];
```

Le tableau ne diminuera, ni ne s'étendra jamais, pas plus qu'il ne contiendra autre chose que des entiers **int**. Les collections sont plus souples. Elles peuvent s'étendre ou se contracter, à mesure que des éléments sont ajoutés ou supprimés. Les tableaux sont destinés à contenir des éléments de type unique, tandis que les collections sont conçues pour contenir des éléments de types différents. Vous pouvez disposer de cette souplesse d'utilisation à l'aide de la conversion boxing, comme suit :

```
ArrayList flexible = new ArrayList( );  
flexible.Add("un"); // Ajoute une chaîne ici  
...  
flexible.Add(99); // et un entier là !
```

Vous ne pouvez pas créer d'instance de tableau avec des éléments en lecture seule. Le code suivant ne pourra pas être compilé.

```
const int[ ] array = {0, 1, 2, 3};  
readonly int[ ] array = {4,2};
```

Toutefois, vous pouvez créer une collection en lecture seule, comme suit :

```
ArrayList flexible = new ArrayList( );  
...  
ArrayList noWrite = ArrayList.ReadOnly(flexible);  
noWrite[0] = 42; // Produit une exception à l'exécution
```

◆ Création de tableaux

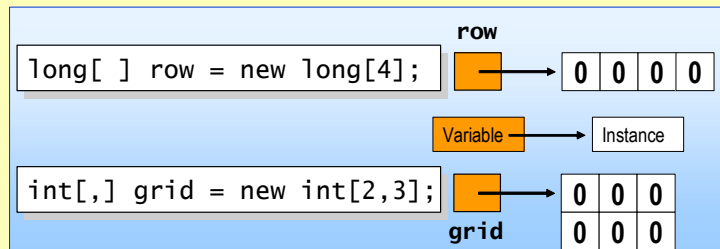
- Création d'instances de tableau
- Initialisation d'éléments de tableau
- Initialisation d'éléments de tableaux multidimensionnels
- Création d'un tableau de taille calculée
- Copie de variables tableau

Dans cette section, vous allez apprendre à créer des instances de tableau, à initialiser explicitement des éléments d'instance de tableau et à copier des variables de tableau.

Création d'instances de tableau

■ La déclaration d'une variable tableau n'entraîne pas la création d'un tableau !

- Vous devez utiliser **new** pour créer explicitement l'instance du tableau
- La valeur implicite par défaut des éléments d'un tableau est zéro



La déclaration d'une variable tableau n'entraîne pas la création effective de l'instance du tableau. La raison en est que les tableaux sont des types référence, et non des types valeur. 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 code suivant entraînera une erreur de compilation :

```
long[] row = new long[];    // Interdit
int[,] grid = new int[];    // Interdit
```

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**. Autrement dit, le code C# :

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

exécutera le code suivant au moment de l'exécution :

```
long[] row = new long[4];
row[0] = 0L;
row[1] = 0L;
row[2] = 0L;
row[3] = 0L;
```

Le compilateur alloue toujours les tableaux en mémoire de manière contiguë, quels que soit le type du tableau et le nombre de dimensions. Si vous créez un tableau avec une expression, telle que `new int[2, 3, 4]`, il est conceptuellement de $2 \times 3 \times 4$, mais l'allocation de mémoire sous-jacente concerne un bloc de mémoire unique suffisamment volumineux pour contenir $2*3*4$ éléments.

Initialisation d'éléments de tableau

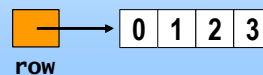
■ Les éléments d'un tableau peuvent être explicitement initialisés

- Vous pouvez utiliser une notation abrégée pratique

```
long[ ] row = new long[4] {0, 1, 2, 3};
```

```
long[ ] row = {0, 1, 2, 3};
```

↑
Équivalent



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]{a, b( ), c*d, e( )+f( )};
```

Vous pouvez également utiliser des initialiseurs de tableau pour initialiser les tableaux de structs :

```
struct Date { ... }
Date[ ] dates = new Date[2];
```

Vous pouvez utiliser cette notation abrégée uniquement lorsque vous initialisez une instance de tableau dans le cadre d'une déclaration de variable tableau et non d'une instruction d'assignation ordinaire.

```
int[ ] data1 = new int[4]{0, 1, 2, 3}; // Autorisé
int[ ] data2 =           {0, 1, 2, 3}; // Autorisé
data2 = new int[4]{0, 1, 2, 3};       // Autorisé
data2 =           {0, 1, 2, 4};       // Interdit
```

En initialisant les tableaux, vous devez explicitement initialiser tous les éléments des tableaux. Il est impossible de laisser les derniers éléments de tableau revenir à leur valeur par défaut de zéro :

```
int[ ] data3 = new int[2]{}; // Interdit
int[ ] data4 = new int[2]{42}; // Interdit
int[ ] data5 = new int[2]{42,42}; // Autorisé
```


Initialisation d'éléments de tableaux multidimensionnels

■ Vous pouvez aussi initialiser des éléments de tableau à plusieurs dimensions

- Tous les éléments doivent être spécifiés

```
int[,] grid = {  
    {5, 4, 3},  
    {2, 1, 0}  
};
```

← Implicitement un nouveau
tableau int[2,3]

✓ 

```
int[,] grid = {  
    {5, 4, 3},  
    {2, 1 }  
};
```

✗

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

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

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

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


Création d'un tableau de taille calculée

- **La taille du tableau n'est pas nécessairement une constante au moment de la compilation**

- Toute expression entière valide fonctionnera
- L'accès aux éléments s'effectue aussi vite dans tous les cas

Taille de tableau spécifiée par une constante entière au moment de la compilation :

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

Taille de tableau spécifiée par une valeur entière au moment de la compilation :

```
string s = Console.ReadLine();  
int size = int.Parse(s);  
long[ ] row = new long[size];
```

Vous pouvez créer des tableaux multidimensionnels à l'aide d'expressions évaluées à l'exécution pour la longueur de chaque dimension, comme le montre le code suivant :

```
System.Console.WriteLine("Entrez le nombre de lignes : ");  
string s1 = System.Console.ReadLine( );  
int rows = int.Parse(s1);  
System.Console.WriteLine("Entrez le nombre de colonnes : ");  
string s2 = System.Console.ReadLine( );  
int cols = int.Parse(s2);  
...  
int[,] matrix = new int[rows,cols];
```

Par ailleurs, vous pouvez utiliser une formule combinant constantes évaluées lors de la compilation et expressions évaluées à l'exécution :

```
System.Console.WriteLine("Entrez le nombre de lignes : ");  
string s1 = System.Console.ReadLine( );  
int rows = int.Parse(s1);  
...  
int[,] matrix = new int[rows,4];
```

Il existe toutefois une restriction mineure. Vous ne pouvez pas utiliser simultanément des expressions évaluées à l'exécution et des initialiseurs de tableaux pour spécifier la taille d'un tableau :

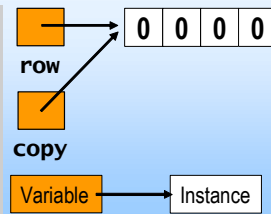
```
string s = System.Console.ReadLine( );  
int size = int.Parse(s);  
int[ ] data = new int[size]{0,1,2,3}; // Interdit
```

Copie de variables tableau

■ La copie d'une variable tableau se contente de copier la variable seulement

- Elle ne copie pas l'instance du tableau
- Deux variables tableau peuvent référencer la même instance de tableau

```
long[ ] row = new long[4];  
long[ ] copy = row;  
...  
row[0]++;  
long value = copy[0];  
Console.WriteLine(value);
```



Lorsque vous copiez une variable tableau, vous n'obtenez pas une copie complète de l'instance du tableau. L'analyse du code de la diapositive révèle ce qui se produit réellement lorsqu'une variable tableau est copiée.

Les instructions suivantes déclarent des variables tableau appelées *copy* et *row* qui font toutes deux référence à la même instance de tableau (de quatre entiers **long**).

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

L'instruction suivante incrémente l'élément initial de cette instance de tableau de 0 à 1. Les deux variables tableau font encore référence à la même instance de tableau, dont l'élément initial est désormais 1.

```
row[0]++;
```

L'instruction suivante initialise un entier **long** appelé *value* à partir de *copy[0]*, qui est l'élément de tableau initial de l'instance de tableau à laquelle *copy* fait référence.

```
long value = copy[0];
```

Étant donné que *copy* et *row* font tous deux référence à la même instance de tableau, l'initialisation de la valeur à partir de *row[0]* produit exactement le même résultat.

L'instruction finale écrit *value* (1) sur la console :

```
Console.WriteLine(value);
```

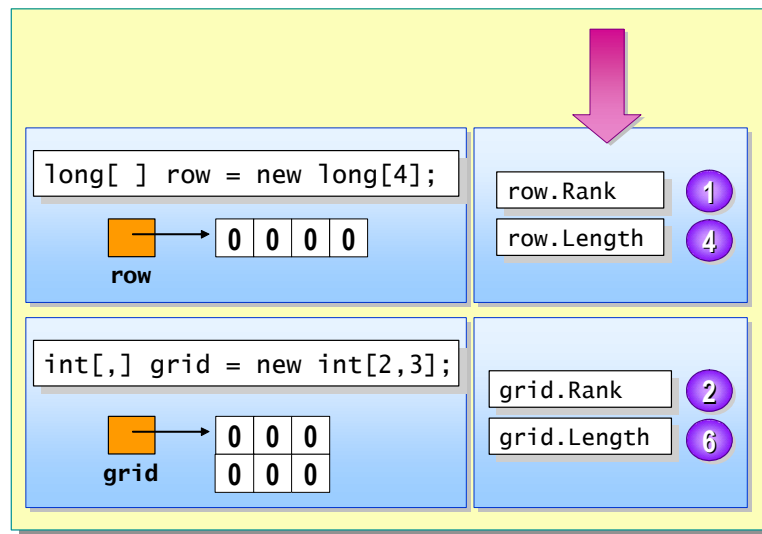
◆ Utilisation des tableaux

- Propriétés des tableaux
- Méthodes de tableau
- Renvoi de tableaux à partir de méthodes
- Passage de tableaux en tant que paramètres
- Arguments de la ligne de commande
- Démonstration : Arguments de la méthode Main
- Utilisation de tableaux avec foreach
- Quiz : Trouver les bogues

Dans cette section, vous allez apprendre à utiliser des tableaux et à passer des tableaux en tant que paramètres à des méthodes.

Vous allez apprendre les règles qui régissent les valeurs par défaut des éléments d'instance de tableau. Les tableaux héritent implicitement de la classe **System.Array** qui fournit de nombreuses propriétés et méthodes. Vous étudierez certaines des propriétés et méthodes communément utilisées. Vous utiliserez l'instruction **foreach** pour parcourir des tableaux par itération. Enfin, vous apprendrez à éviter certains écueils traditionnels.

Propriétés des tableaux



La propriété **Rank** est une valeur entière en lecture seule qui spécifie la dimension de l'instance du tableau. Par exemple :

```
int[ ] one = new int[a];  
int[,] two = new int[a,b];  
int[, ,] three = new int[a,b,c];
```

les valeurs de rang résultantes sont les suivantes :

```
one.Rank == 1  
two.Rank == 2  
three.Rank == 3
```

La propriété **Length** est une valeur entière en lecture seule qui spécifie la longueur totale de l'instance du tableau. Par exemple, les trois déclarations de tableau ci-dessus produisent les valeurs suivantes :

```
one.Length == a  
two.Length == a * b  
three.Length == a * b * c
```

Méthodes de tableau

■ Méthodes couramment utilisées

- **Sort** – Trie les éléments d'un tableau de rang 1
- **Clear** – Initialise une plage d'éléments à zéro ou **null**
- **Clone** – Crée une copie du tableau
- **GetLength** – Retourne la longueur d'une dimension donnée
- **IndexOf** – Retourne l'index de la première occurrence d'une valeur

La classe **System.Array** (une classe implicitement prise en charge par tous les tableaux) fournit de nombreuses méthodes que vous pouvez utiliser lorsque vous travaillez avec les tableaux. Cette rubrique décrit certaines des méthodes les plus communément utilisées.

■ Méthode **Sort**

Cette méthode effectue un tri dans un tableau fourni en tant qu'argument. Vous pouvez utiliser cette méthode pour trier les tableaux de structures et de classes tant qu'ils prennent en charge l'interface **IComparable**.

```
int[ ] data = {4,6,3,8,9,3}; // Non trié
System.Array.Sort(data);    // Trié
```

■ Méthode **Clear**

Cette méthode réinitialise une plage d'éléments de tableau à zéro (pour les types valeur) ou **null** (pour les types référence), comme suit :

```
int[ ] data = {4,6,3,8,9,3};
System.Array.Clear(data, 0, data.Length);
```

■ Méthode **Clone**

Cette méthode crée une nouvelle instance de tableau dont les éléments sont des copies des éléments du tableau cloné. Vous pouvez utiliser cette méthode pour cloner des tableaux de structs et de classes définis par l'utilisateur. Vous trouverez ci-dessous un exemple :

```
int[ ] data = {4,6,3,8,9,3};  
int[ ] clone = (int[ ])data.Clone( );
```

Attention La méthode **Clone** effectue une copie partielle. Si le tableau copié contient des références aux objets, ce sont les références qui seront copiées, et non les objets, et les deux tableaux feront référence aux mêmes objets.

■ Méthode **GetLength**

Cette méthode renvoie la longueur d'une dimension fournie en argument entier. Vous pouvez utiliser cette méthode pour vérifier les limites des tableaux multidimensionnels. Vous trouverez ci-dessous un exemple :

```
int[,] data = { {0, 1, 2, 3}, {4, 5, 6, 7} };  
int dim0 = data.GetLength(0); // == 2  
int dim1 = data.GetLength(1); // == 4
```

■ Méthode **IndexOf**

Cette méthode renvoie l'index entier de la première occurrence d'une valeur en argument, ou -1 si la valeur n'est pas présente. Vous ne pouvez utiliser cette méthode que sur les tableaux à une dimension. Vous trouverez ci-dessous un exemple :

```
int[ ] data = {4,6,3,8,9,3};  
int where = System.Array.IndexOf(data, 9); // == 4
```

Remarque Suivant le type des éléments du tableau, la méthode **IndexOf** peut nécessiter que vous passiez outre la méthode **Equals** pour le type d'élément. Vous en apprendrez plus à ce sujet dans un module ultérieur.

Renvoi de tableaux à partir de méthodes

- Vous pouvez déclarer des méthodes qui retournent des tableaux

```
class Example {  
    static void Main( ) {  
        int[ ] array = CreateArray(42);  
        ...  
    }  
    static int[ ] CreateArray(int size) {  
        int[ ] created = new int[size];  
        return created;  
    }  
}
```

Dans la diapositive, la méthode **CreateArray** est implémentée à l'aide de deux instructions. Vous pouvez combiner ces deux instructions dans une instruction **return**, comme suit :

```
static int[ ] CreateArray(int size) {  
    return new int[size];  
}
```

Les programmeurs C++ noteront que, dans les deux cas, la taille du tableau renvoyé n'est pas spécifiée. Si vous spécifiez la taille du tableau, vous obtenez une erreur de compilation, comme dans l'exemple suivant :

```
static int[4] CreateArray( ) // erreur du compilateur  
  
{  
    return new int[4];  
}
```

Vous pouvez également renvoyer des tableaux de rang supérieur à un, comme dans l'exemple suivant :

```
static int[,] CreateArray( ) {  
    string s1 = System.Console.ReadLine( );  
    int rows = int.Parse(s1);  
    string s2 = System.Console.ReadLine( );  
    int cols = int.Parse(s2);  
    return new int[rows,cols];  
}
```

Passage de tableaux en tant que paramètres

- **Un paramètre tableau est une copie de la variable tableau**

- Ce n'est pas une copie de l'instance du tableau

```
class Example2 {  
    static void Main( ) {  
        int[ ] arg = {10, 9, 8, 7};  
        Method(arg);  
        System.Console.WriteLine(arg[0]);  
    }  
    static void Method(int[ ] parameter) {  
        parameter[0]++;  
    }  
}
```

Cette méthode modifie
l'instance du tableau
d'origine créée dans Main

Lorsque vous passez à une méthode une variable tableau en tant qu'argument, le paramètre de la méthode devient une copie de l'argument de la variable tableau. Autrement dit, le paramètre tableau est initialisé à partir de l'argument. Vous utilisez la même syntaxe pour initialiser le paramètre tableau ou pour initialiser une variable tableau, comme décrit précédemment dans la rubrique Copie de variables tableau. L'argument tableau et le paramètre tableau font tous deux référence à la même instance de tableau.

Dans le code de la diapositive, **arg** est initialisé avec une instance de tableau de longueur 4 qui contient les entiers 10, 9, 8 et 7. **Arg** est ensuite passé en tant qu'argument à **Method**. **Method** accepte **arg** en tant que paramètre, ce qui signifie que **arg** et le paramètre font tous deux référence à la même instance de tableau (celle utilisée pour initialiser **arg**). L'expression `parameter[0]++` dans **Method** incrémente ensuite l'élément initial de la même instance de tableau de 10 à 11. (Étant donné qu'on accède à l'élément initial d'un tableau en spécifiant la valeur d'index 0 et non 1, l'élément initial est également appelé l'élément « zéro ».) **Method** renvoie et **Main** écrit ensuite la valeur de `arg[0]` sur la console. Le paramètre **arg** faisant encore référence à la même instance de tableau, et l'élément zéro de cette instance de tableau venant juste d'être incrémenté, la valeur 11 est écrite sur la console.

Étant donné que le passage d'une variable tableau ne crée pas de copie complète de l'instance du tableau, le passage d'un tableau en tant que paramètre est très rapide. Si vous voulez une méthode qui dispose d'un accès en écriture pour l'instance de tableau de l'argument, ce comportement de copie partielle convient parfaitement.

La méthode **Array.Copy** est utile si vous souhaitez garantir que la méthode appelée ne modifiera pas l'instance de tableau et que vous êtes prêt à accepter un temps d'exécution plus long en échange de cette garantie. Vous pouvez également passer un tableau récemment créé en tant que paramètre tableau, comme suit :

```
Method(new int[4]{10, 9, 8, 7});
```

Arguments de la ligne de commande

- **Le runtime passe des arguments de ligne de commande à Main**

- **Main** peut accepter un tableau de chaînes comme paramètre
- Le nom du programme n'est pas un membre du tableau

```
class Example3 {  
    static void Main(string[] args) {  
        for (int i = 0; i < args.Length; i++) {  
            System.Console.WriteLine(args[i]);  
        }  
    }  
}
```

Lorsque vous exécutez des programmes sur la consoles, vous passez souvent des arguments supplémentaires dans la ligne de commande. Par exemple, si vous exécutez le programme **pkzip** en invite de commandes, vous pouvez spécifier des arguments supplémentaires pour contrôler la création des fichiers .zip. La commande suivante ajoute de manière récursive tous les fichiers de code *.cs dans code.zip :

```
C:\> pkzip -add -rec -path=relative c:\code *.cs
```

Si vous aviez écrit le programme **pkzip** en C#, vous pourriez capturer ces arguments de ligne de commande sous la forme d'un tableau de chaînes que le runtime passerait à **Main** :

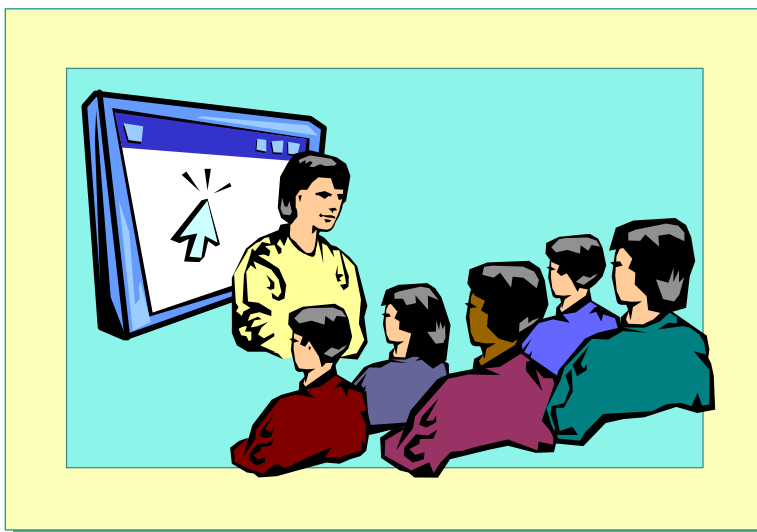
```
class PKZip {  
    static void Main(string[] args) {  
        ...  
    }  
}
```

Dans cet exemple, lorsque vous exécutez le programme **pkzip**, le runtime exécute en fait le code suivant :

```
string[] args = {  
    "-add",  
    "-rec",  
    "-path=relative",  
    "c:\\code",  
    "*.cs"  
};  
PKZip.Main(args);
```

Remarque Contrairement aux langages C et C++, le nom du programme lui-même n'est pas passé comme args[0] en C#.

Démonstration : Arguments de la méthode Main



Dans cette démonstration, vous allez découvrir comment passer des arguments de ligne de commande à un programme C#.

Utilisation de tableaux avec foreach

- L'instruction **foreach** simplifie de nombreux aspects de la gestion des tableaux

```
class Example4 {  
    static void Main(string[] args) {  
        foreach (string arg in args) {  
            System.Console.WriteLine(arg);  
        }  
    }  
}
```

Lorsqu'elle est applicable, l'instruction **foreach** est utile car elle permet de faire l'économie du mécanisme d'itération sur chaque élément d'un tableau. Sans **foreach**, vous écririez :

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

Avec **foreach**, vous pouvez écrire :

```
foreach (string arg in args) {  
    System.Console.WriteLine(arg);  
}
```

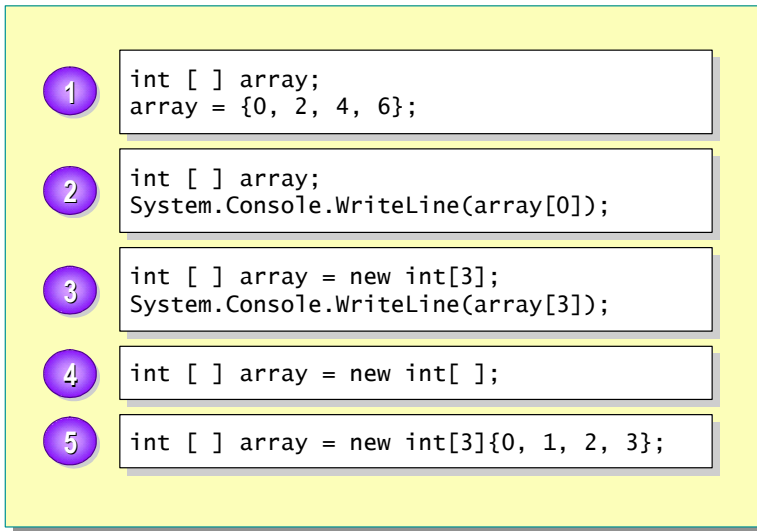
Notez qu'avec l'instruction **foreach**, les éléments suivants sont inutiles :

- index entier (`int i`)
- contrôle des limites du tableau (`i < args.Length`)
- expression d'accès au tableau (`args[i]`)

Vous pouvez également utiliser l'instruction **foreach** pour parcourir par itération les éléments d'un tableau de rang 2 ou supérieur. Par exemple, l'instruction **foreach** suivante écrit les valeurs 0, 1, 2, 3, 4 et 5 :

```
int[,] numbers = { {0,1,2}, {3,4,5} };  
foreach (int number in numbers) {  
    System.Console.WriteLine(number);  
}
```

Quiz : Trouver les bogues

A yellow rectangular box with a thin grey border and a slight drop shadow. Inside the box, there are five numbered items. Each item consists of a purple circular icon with a white number (1 through 5) and a white rectangular text box containing C# code. The items are arranged vertically.

- 1 `int [] array;
array = {0, 2, 4, 6};`
- 2 `int [] array;
System.Console.WriteLine(array[0]);`
- 3 `int [] array = new int[3];
System.Console.WriteLine(array[3]);`
- 4 `int [] array = new int[];`
- 5 `int [] array = new int[3]{0, 1, 2, 3};`

Vous pouvez travailler avec un partenaire pour trouver les bogues du code reproduit sur la diapositive. Les réponses se trouvent à la page suivante.

Réponses

Voici les bogues que vous devez avoir trouvés :

1. Un initialiseur de tableau est utilisé dans une assignation sans expression de création de tableau. Le raccourci `int[] array = { ... };` n'est autorisé que dans une déclaration de tableau. Ce bogue produira une erreur de compilation.
2. La variable tableau a été déclarée, mais il n'y a aucune expression de tableau, donc aucune instance de tableau. Ce bogue produira également une erreur de compilation.
3. Une erreur classique de décalage d'une unité hors limites. Le tableau a une longueur de trois, ce qui autorise les valeurs d'index 0, 1 et 2. Souvenez-vous que les tableaux sont indexés à partir de zéro en C#. Ce bogue produira une exception d'exécution **System.IndexOutOfRangeException**.
4. La longueur du tableau n'est pas spécifiée dans l'expression de création du tableau. La longueur d'un tableau doit être spécifiée lors de la création de l'instance de tableau.
5. Le nombre d'éléments de tableau est spécifié comme étant de 3 dans `new int[3]` mais il y a quatre littéraux entiers dans l'initialiseur de tableau.

Atelier 6.1 : Création et utilisation de tableaux



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer et utiliser des tableaux de types valeur ;
- passer des arguments à **Main** ;
- créer et utiliser des tableaux de taille calculée ;
- utiliser des tableaux de rangs multiples.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les éléments suivants :

- utiliser les instructions de programmation C# ;
- écrire et utiliser des méthodes en C#.

Durée approximative de cet atelier : 60 minutes

Exercice 1

Utilisation d'un tableau de types valeur

Dans cet exercice, vous allez écrire un programme qui attend le nom d'un fichier texte en tant qu'argument de **Main**. Le programme résumera le contenu du fichier texte. Il lira le contenu du fichier texte dans un tableau de caractères, puis parcourera le tableau de manière itérative, en comptant le nombre de voyelles et de consonnes. Enfin, il affichera le nombre total de caractères, de voyelles, de consonnes et de sauts de lignes dans la console.

► **Pour capturer le nom du fichier texte comme paramètre de la méthode Main**

1. Ouvrez le projet FileDetails.sln. Ce projet se trouve dans *dossier d'installation*\Labs\Lab06\Starter\FileDetails.
2. Ajoutez un tableau de chaînes appelé **args** comme paramètre de la méthode **Main** de la classe **FileDetails**. Ce tableau contiendra tous les arguments de ligne de commande fournis lors de l'exécution du programme. C'est de cette manière que le runtime passe des arguments de ligne de commande à **Main**. Dans cet exercice, l'argument de ligne de commande passé à **Main** sera le nom du fichier texte.
3. Ajoutez une instruction à **Main** qui écrit la longueur de **args** sur la console. Cette instruction vérifiera que la longueur de **args** est égale à zéro lorsque aucun argument de ligne de commande n'est passé à **Main** par le runtime.
4. Ajoutez une instruction **foreach** à **Main** qui écrit chaque chaîne du tableau **args** sur la console. Cette instruction vérifiera que **Main** reçoit les arguments de ligne de commande envoyés par le runtime.

Votre code terminé doit se présenter comme suit :

```
static void Main(string[] args)
{
    Console.WriteLine(args.Length);
    foreach (string arg in args) {
        Console.WriteLine(arg);
    }
}
```

5. Compilez le programme FileDetails.cs et corrigez les erreurs, le cas échéant. Exécutez le programme à partir de la ligne de commande, en ne fournissant aucun argument de ligne de commande. Vérifiez que la longueur du tableau **args** est de zéro.

Conseil Pour exécuter le programme à partir de la ligne de commande, ouvrez la fenêtre Commande et accédez au dossier *dossier d'installation*\Labs\Lab06\Starter\FileDetails\bin\Debug. Le fichier exécutable se trouve dans ce dossier.

6. Exécutez le programme à partir de la ligne de commande, en spécifiant le nom du fichier *dossier d'installation\Labs\Lab06\Solution\FileDetails\FileDetails.cs*. Vérifiez que le runtime passe le nom de fichier à **Main**.
7. Testez le programme en spécifiant différents arguments de ligne de commande, et vérifiez que chacun d'entre eux est écrit sur la console. Mettez en commentaire les instructions qui écrivent sur la console.
8. Ajoutez une instruction dans **Main** qui déclare une variable **string** appelée *fileName* et initialisez-la avec `args[0]`.

► **Pour lire le contenu du fichier texte dans un tableau**

1. Supprimez le commentaire des déclarations et d'initialisations de `FileStream` et `StreamReader`.
2. Déterminez la longueur du fichier texte.

Conseil Pour repérer une propriété appropriée de la classe **Stream**, faites une recherche sur la classe `Stream` dans l'aide du kit de développement .NET Framework SDK.

3. Ajoutez une instruction dans **Main** qui déclare une variable tableau de caractères appelée *contents*. Initialisez *contents* avec une nouvelle instance de tableau de même longueur que le fichier texte, que vous venez de déterminer.
4. Ajoutez une instruction **for** dans **Main**. Le corps de l'instruction **for** lira un caractère depuis *reader* et l'ajoutera dans *contents*.

Conseil Utilisez la méthode **Read**, qui ne prend aucun paramètre et renvoie **int**. Placez le résultat dans un **char** avant de le stocker dans le tableau.

5. Ajoutez une instruction **foreach** dans **Main** qui écrit l'ensemble du tableau de caractères sur la console caractère par caractère. Cette instruction vérifiera que le fichier texte a été correctement lu dans le tableau *contents*.

Votre code terminé doit se présenter comme suit :

```
static void Main(string[ ] args)
{
    string fileName = args[0];
    FileStream stream = new FileStream(fileName,
    ↵                               FileMode.Open);
    StreamReader reader = new StreamReader(stream);
    int size = (int)stream.Length;
    char[ ] contents = new char[size];
    for (int i = 0; i < size; i++) {
        contents[i] = (char)reader.Read( );
    }
    foreach(char ch in contents) {
        Console.Write(ch);
    }
}
```

6. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme en spécifiant le nom du fichier *dossier d'installation\Labs\Lab06\Solution\FileDetails\FileDetails.cs* comme argument de ligne de commande. Vérifiez que le contenu du fichier est correctement écrit sur la console.
7. Mettez en commentaire l'instruction **foreach**.
8. Fermez l'objet **Reader** en appelant la méthode **StreamReader** appropriée.

► Pour classer et résumer le contenu du fichier

1. Déclarez une nouvelle méthode statique appelée **Summarize** dans la classe **FileDetails**. Cette méthode ne renvoie rien et attend en paramètre un tableau de caractères. Ajoutez une instruction dans **Main** qui appelle la méthode **Summarize**, en passant *contents* comme argument.
2. Ajoutez une instruction **foreach** dans **Summarize** qui étudie chaque caractère de l'argument tableau. Comptez les voyelles, consonnes et sauts de ligne qui se créent, et stockez les résultats dans des variables distinctes.

Conseil Pour déterminer si un caractère est une voyelle, créez une chaîne contenant toutes les voyelles possibles, puis appliquez la méthode **IndexOf** sur cette chaîne afin de déterminer si le caractère existe dans cette chaîne :

```
if ("AEIOUYaeiouy".IndexOf(myCharacter) != -1) {
    // myCharacter est une voyelle
} else {
    // myCharacter n'est pas une voyelle
}
```

3. Écrivez quatre lignes sur la console afin d'afficher :

- le nombre total de caractères du fichier ;
- le nombre total de voyelles du fichier ;
- le nombre total de consonnes du fichier ;
- le nombre total de lignes dans le fichier.

Votre code terminé doit se présenter comme suit :

```
static void Summarize(char[ ] contents)
{
    int vowels = 0, consonants = 0, lines = 0;
    foreach (char current in contents) {
        if (Char.IsLetter(current)) {
            if ("AEIOUYaeiouy".IndexOf(current) != -1)
            {
                vowels++;
            } else {
                consonants++;
            }
        }
        else if (current == '\n') {
            lines++;
        }
    }
    Console.WriteLine("Nombre total de caractères : 
↪{0}", contents.Length);
    Console.WriteLine("Nombre total de voyelles : 
↪{0}", vowels);
    Console.WriteLine("Nombre total de consonnes : 
↪{0}", consonants);
    Console.WriteLine("Nombre total de lignes : 
↪{0}", lines);
}
```

4. Compilez le programme et corrigez les erreurs, le cas échéant.
Exécutez le programme à partir de la ligne de commande afin de résumer le contenu du fichier de solution : *dossier d'installation*\Labs\Lab06\Solution\FileDetails\FileDetails.cs.
Les totaux corrects sont les suivants :
 - 1 379 caractères
 - 257 voyelles
 - 403 consonnes
 - 41 lignes

Exercice 2

Multiplication de matrices

Dans cet exercice, vous allez écrire un programme qui utilise des tableaux pour multiplier des matrices entre elles. Le programme lira quatre valeurs entières à partir de la console et les stockera dans une matrice d'entiers de 2 x 2. Il lira ensuite quatre valeurs entières supplémentaires à partir de la console et les stockera dans une deuxième matrice d'entiers de 2 x 2. Le programme multipliera ensuite les deux matrices ensemble, stockant le résultat dans une troisième matrice d'entiers de 2 x 2. Enfin, il affichera la matrice résultante sur la console.

La formule de multiplication de deux matrices, A et B, est la suivante :

$$\begin{array}{cc} A1 & A2 \\ A3 & A4 \end{array} \times \begin{array}{cc} B1 & B2 \\ B3 & B4 \end{array} = \begin{array}{cc} A1.B1 + A2.B3 & A1.B2 + A2.B4 \\ A3.B1 + A4.B3 & A3.B2 + A4.B4 \end{array}$$

► Pour multiplier deux matrices ensemble

1. Ouvrez le projet MatrixMultiply.sln dans le dossier *dossier d'installation*\Labs\Lab06\Starter\MatrixMultiply.
2. Dans la classe **MatrixMultiply**, ajoutez une instruction dans **Main** qui déclare un tableau de 2 x 2 entiers **int** et nomme le tableau **a**. La solution finale du programme lira les valeurs du tableau **a** à partir de la console. Pour l'instant, initialisez **a** avec les valeurs de la table suivante. (Cela permet de vérifier que la multiplication est effectuée correctement et que le processus de refactorisation qui suit retient le comportement prévu.)

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

3. Ajoutez une instruction dans **Main** qui déclare un tableau de 2 x 2 entiers **int** et nomme le tableau **b**. La solution finale du programme lira les valeurs du tableau **b** à partir de la console. Pour l'instant, initialisez **b** avec les valeurs de la table suivante :

$$\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

4. Ajoutez une instruction dans **Main** qui déclare un tableau de 2 x 2 entiers **int** et nomme le tableau **result**. Initialisez **result** avec les formules de cellule suivantes :

$$\begin{array}{cc} a[0,0] * b[0,0] + a[0,1] * b[1,0] & a[0,0] * b[0,1] + a[0,1] * b[1,1] \\ a[1,0] * b[0,0] + a[1,1] * b[1,0] & a[1,0] * b[0,1] + a[1,1] * b[1,1] \end{array}$$

5. Ajoutez quatre instructions dans **Main** qui écrivent les quatre valeurs **int** du tableau **result** sur la console. Ces instructions vous permettront de vérifier que vous avez correctement copié les formules.

Votre code terminé doit se présenter comme suit :

```
static void Main( )
{
    int[,] a = new int[2,2];
    a[0,0] = 1; a[0,1] = 2;
    a[1,0] = 3; a[1,1] = 4;

    int[,] b = new int[2,2];
    b[0,0] = 5; b[0,1] = 6;
    b[1,0] = 7; b[1,1] = 8;

    int[,] result = new int[2,2];
    result[0,0]=a[0,0]*b[0,0] + a[0,1]*b[1,0];
    result[0,1]=a[0,0]*b[0,1] + a[0,1]*b[1,1];
    result[1,0]=a[1,0]*b[0,0] + a[1,1]*b[1,0];
    result[1,1]=a[1,0]*b[0,1] + a[1,1]*b[1,1];

    Console.WriteLine(result[0,0]);
    Console.WriteLine(result[0,1]);
    Console.WriteLine(result[1,0]);
    Console.WriteLine(result[1,1]);
}
```

6. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme. Vérifiez que les quatre valeurs de **result** sont les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

► **Pour sortir le résultat à l'aide d'une méthode utilisant un paramètre tableau**

1. Déclarez une nouvelle méthode statique appelée **Output** dans la classe **MatrixMultiply**. Cette méthode ne renvoie rien et attend comme paramètre un tableau **int** de rang 2 appelé **result**.
2. Coupez dans **Main** les quatre instructions qui écrivent les quatre valeurs de **result** sur la console et collez-les dans **Output**.
3. Ajoutez une instruction dans **Main** qui appelle la méthode **Output**, en passant **result** comme argument. (Cela doit remplacer le code qui a été coupé dans l'étape précédente.)

Votre code terminé doit se présenter comme suit :

```
static void Output(int[,] result)
{
    Console.WriteLine(result[0,0]);
    Console.WriteLine(result[0,1]);
    Console.WriteLine(result[1,0]);
    Console.WriteLine(result[1,1]);
}
```

4. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme. Vérifiez que les quatre valeurs écrites sur la console sont encore les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

5. Retraavaillez la méthode **Output** de manière à utiliser deux instructions **for** imbriquées au lieu de quatre instructions **WriteLine**. Utilisez la valeur littérale 2 dans les deux contrôles de limites de tableau.

Votre code terminé doit se présenter comme suit :

```
static void Output(int[,] result)
{
    for (int r = 0; r < 2; r++) {
        for (int c = 0; c < 2; c++) {
            Console.Write("{0} ", result[r,c]);
        }
        Console.WriteLine( );
    }
}
```

6. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme. Vérifiez que les quatre valeurs écrites sur la console sont encore les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

7. Modifiez à nouveau la méthode **Output**, afin de la rendre plus générique. Remplacez la valeur littérale 2 dans les contrôles de limites de tableau par des appels à la méthode **GetLength** de chacun des deux contrôles.

Votre code terminé doit se présenter comme suit :

```
static void Output(int[,] result)
{
    for (int r = 0; r < result.GetLength(0); r++) {
        for (int c = 0; c < result.GetLength(1); c++) {
            Console.Write("{0} ", result[r,c]);
        }
        Console.WriteLine( );
    }
}
```

8. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme. Vérifiez que les quatre valeurs écrites sur la console sont encore les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

► **Pour calculer le résultat dans une méthode et le renvoyer**

1. Déclarez une nouvelle méthode statique appelée **Multiply** au sein de la classe **MatrixMultiply**. Cette méthode renverra un tableau **int** de rang 2 et attendra deux tableaux **int** de rang 2, respectivement nommés **a** et **b**, en tant que paramètres.
2. Copiez (mais ne coupez pas) la déclaration et l'initialisation de **result** de **Main** dans **Multiply**.
3. Ajoutez une instruction **return** dans **Multiply** qui renvoie **result**.
4. Remplacez l'initialisation de **result** dans **Main** par un appel à **Multiply**, en passant **a** et **b** comme arguments.

Votre code terminé doit se présenter comme suit :

```
static int[,] Multiply(int[,] a, int [,] b)
{
    int[,] result = new int[2,2];
    result[0,0]=a[0,0]*b[0,0] + a[0,1]*b[1,0];
    result[0,1]=a[0,0]*b[0,1] + a[0,1]*b[1,1];
    result[1,0]=a[1,0]*b[0,0] + a[1,1]*b[1,0];
    result[1,1]=a[1,0]*b[0,1] + a[1,1]*b[1,1];
    return result;
}
```

5. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme. Vérifiez que les quatre valeurs écrites sur la console sont encore les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

► **Pour calculer le résultat dans une méthode à l'aide d'instructions for**

1. Remplacez l'initialisation de **result** dans **Multiply** par un nouveau tableau d'**ints** de 2 x 2.
2. Ajoutez deux instructions **for** imbriquées dans **Multiply**. Utilisez un entier appelé *r* dans l'instruction **for** externe pour parcourir chaque index de la première dimension de **result**. Utilisez un entier appelé *c* dans l'instruction **for** interne pour parcourir chaque index de la deuxième dimension de **result**. Utilisez la valeur littérale 2 pour les deux limites de tableau. Le corps de l'instruction **for** interne devra calculer et définir la valeur de **result[r,c]** à l'aide de la formule suivante :

```
result[r,c] = a[r,0] * b[0,c]
              + a[r,1] * b[1,c]
```


Votre code terminé doit se présenter comme suit :

```
static int[,] Multiply(int[,] a, int [,] b)
{
    int[,] result = new int[2,2];
    for (int r = 0; r < 2; r++) {
        for (int c = 0; c < 2; c++) {
            result[r,c] += a[r,0] * b[0,c] + a[r,1] * b[1,c];
        }
    }
    return result;
}
```

3. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme. Vérifiez que les quatre valeurs écrites sur la console sont encore les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

► **Pour entrer la première matrice à partir de la console**

1. Remplacez l'initialisation de **a** dans **Main** par un nouveau tableau d'**ints** de 2 x 2.
2. Ajoutez dans **Main** des instructions qui demandent à l'utilisateur les quatre valeurs de **a** et les lisent à partir de la console. Ces instructions doivent être placées avant l'appel de la méthode **Multiply**. Les instructions qui lisent *une* valeur depuis la console sont les suivantes :

```
string s = Console.ReadLine( );
a[0,0] = int.Parse(s);
```

3. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme, en entrant les même quatre valeurs pour **a** à partir de la console (c'est-à-dire 1, 2, 3 et 4). Vérifiez que les quatre valeurs écrites sur la console sont encore les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

4. Déclarez une nouvelle méthode statique appelée **Input** au sein de la classe **MatrixMultiply**. Cette méthode ne renvoie rien et attend comme paramètre un tableau **int** de rang 2 appelé **dst**.
5. Coupez les instructions qui lisent quatre valeurs dans **a** à partir de **Main** et collez-les dans **Input**. Ajoutez dans **Main** une instruction qui appelle **Input**, passant **a** comme paramètre. Cette instruction doit être placée avant l'appel à **Multiply**.

6. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme, en entrant les même quatre valeurs pour **a** à partir de la console (c'est-à-dire 1, 2, 3 et 4). Vérifiez que les quatre valeurs écrites sur la console sont encore les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

7. Changez la méthode **Input** de manière à utiliser deux instructions **for** imbriquées. Utilisez la valeur littérale 2 pour les deux limites de tableau. Insérez une instruction **Write** dans la méthode **Input** afin de demander à l'utilisateur chaque entrée.

Votre code terminé doit se présenter comme suit :

```
static void Input(int[,] dst)
{
    for (int r = 0; r < 2; r++) {
        for (int c = 0; c < 2; c++) {
            Console.Write(
↳ "Entrer la valeur de [{0},{1}] : ", r, c);
            string s = Console.ReadLine();
            dst[r,c] = int.Parse(s);
        }
    }
    Console.WriteLine();
}
```

8. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme, en entrant les même quatre valeurs pour **a** à partir de la console (c'est-à-dire 1, 2, 3 et 4). Vérifiez que les quatre valeurs écrites sur la console sont encore les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

► Pour entrer la deuxième matrice de la console

1. Remplacez l'initialisation de **b** dans **Main** par un nouveau tableau d'**ints** de 2 x 2 dont les quatre valeurs sont égales à zéro par défaut.
2. Ajoutez une instruction dans **Main** qui lise les valeurs dans **b** à partir de la console en appelant la méthode **Input** et en passant **b** comme l'argument.
3. Compilez le programme et corrigez les erreurs, le cas échéant. Exécutez le programme, en entrant les mêmes quatre valeurs pour **a** (1, 2, 3 et 4) et les mêmes quatre valeurs pour **b** (5, 6, 7 et 8). Vérifiez que les quatre valeurs écrites sur la console sont encore les suivantes :

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

4. Exécutez le programme avec d'autres données. Comparez vos résultats avec ceux d'un autre stagiaire pour voir si vous obtenez les mêmes résultats pour la même entrée.

Contrôle des acquis

- Vue d'ensemble des tableaux
- Création de tableaux
- Utilisation des tableaux

1. Déclarez un tableau d'entiers **int** de rang 1 appelé *evens* et initialisez-le avec les cinq premiers nombres pairs, en commençant par zéro.
2. Écrivez une instruction qui déclare une variable appelée *crowd* de type **int**, et initialisez-la avec le deuxième élément de *evens*. Souvenez-vous que le deuxième élément ne réside pas dans l'index 2 car les index de tableau ne démarrent pas à 1.
3. Écrivez deux instructions. La première déclarera un tableau d'entiers **int** de rang 1 appelé *copy* ; la deuxième assignera *evens* à *copy*.

4. Écrivez une méthode statique appelée **Method** qui renvoie un tableau d'entiers **int** de rang 2 et n'attend aucun argument. Le corps de **Method** contiendra une seule instruction **return**. Cette instruction renvoie un nouveau tableau de rang 2 de dimensions 3 et 5 dont les 15 éléments sont tous initialisés à 42.

5. Écrivez une méthode statique appelée **Parameter** qui ne renvoie rien et attend un tableau à deux dimensions comme seul argument. Le corps de la méthode contiendra deux instructions **WriteLine** qui écrivent la longueur de chaque dimension sur la console.

6. Écrivez une instruction **foreach** qui parcourt un tableau de chaînes à une dimension appelé *names*, en écrivant chaque nom sur la console.

Module 7 : Notions fondamentales de la programmation orientée objet

Table des matières

Vue d'ensemble	1
Classes et objets	2
Utilisation de l'encapsulation	10
C# et orientation objet	21
Atelier 7.1 : Création et utilisation de classes	39
Définition de systèmes orientés objet	52
Contrôle des acquis	61



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- **Classes et objets**
- **Utilisation de l'encapsulation**
- **C# et orientation objet**
- **Définition de systèmes orientés objet**

C# est un langage de programmation orienté objet. Dans cette leçon, vous allez apprendre la terminologie et les concepts requis pour créer et utiliser des classes en C#.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- définir les termes *objet* et *classe* dans le contexte de la programmation orientée objet
- décrire les trois principales caractéristiques d'un objet : l'identité, l'état et le comportement ;
- décrire l'abstraction et la façon dont elle permet de créer des classes réutilisables faciles à gérer ;
- utiliser l'encapsulation pour associer des méthodes et des données dans une seule classe et appliquer l'abstraction ;
- expliquer les concepts d'héritage et de polymorphisme ;
- créer et utiliser des classes en C#.

◆ Classes et objets

- Qu'est-ce qu'une classe ?
- Qu'est-ce qu'un objet ?
- Comparaison entre classes et structs
- Abstraction

Toute la structure de C# repose sur le modèle de programmation orienté objet. Pour faire le meilleur usage de C# en tant que langage, vous devez comprendre la nature de la programmation orientée objet.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- définir les termes *objet* et *classe* dans le cadre de la programmation orientée objet ;
- expliquer le concept d'abstraction.

Qu'est-ce qu'une classe ?

■ Pour le philosophe...

- Un artefact de la *classification* humaine !
- Une *classification* en fonction d'un comportement ou d'attributs communs
- Un accord portant sur les descriptions et les noms des *classes* utiles
- La création d'un vocabulaire ; nous communiquons ; nous pensons !



■ Pour le programmeur orienté objet...

- Une construction syntaxique nommée qui décrit un comportement et des attributs communs
- Une structure de données qui inclut les données et les fonctions

La racine du mot classification est *classe*. Classifier consiste à constituer des classes. C'est une activité très naturelle, qui n'est pas réservée aux programmeurs ! Par 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*. Les phrases manqueraient singulièrement de concision. En fait, la communication serait probablement impossible. Tant que tous acceptent la signification d'un mot, c'est-à-dire tant que nous parlons tous la même langue, la communication fonctionne bien, et nous pouvons exprimer de façon concise des idées complexes mais précises. Nous utilisons ensuite ces concepts nommés pour former des concepts de niveau supérieur et accroître le pouvoir d'expression de la communication.

Tous les langages de programmation peuvent décrire des données et des fonctions communes. Cette capacité à décrire des fonctionnalités communes aide à éviter la duplication. L'une des principales devises de la programmation est « Ne te répète pas ». Un code dupliqué pose problème, car il est plus difficile à tenir à jour. Un code qui ne se répète pas est plus facile à tenir à jour, notamment car il y en a moins ! Les langages orientés objet vont encore plus loin en autorisant les descriptions de classes (ensembles d'objets) qui partagent une structure et un comportement. S'il est réalisé correctement, ce paradigme fonctionne parfaitement et s'insère naturellement dans le mode de raisonnement et de communication des gens.

Les classes ne sont pas limitées à la classification d'objets concrets (comme les voitures) ; elles peuvent aussi servir à classer des concepts abstraits (comme le temps). Toutefois, lorsque vous classifiez des concepts abstraits, les frontières sont moins claires et la qualité de la conception prend encore plus d'importance.

La seule exigence à avoir à l'égard d'une classe est qu'elle facilite la communication.

Qu'est-ce qu'un objet ?

- Un objet est une instance d'une classe
- Les objets ont :
 - Une identité : ils sont reconnaissables les uns des autres
 - Un comportement : ils peuvent réaliser des tâches
 - Un état : ils stockent des informations



Le mot *voiture* a différentes significations selon le contexte. Nous utilisons parfois le mot *voiture* pour évoquer le concept général de la voiture : nous parlons de *voiture* en tant que *classe*, ce qui signifie l'ensemble de toutes les voitures, sans penser à aucune en particulier. À d'autres moments, nous utilisons le mot *voiture* pour évoquer une voiture spécifique. Les programmeurs emploient le terme *objet* ou *instance* pour faire référence à une voiture spécifique. Il est important de comprendre cette différence.

Les trois caractéristiques que sont l'identité, le comportement et l'état constituent un moyen pratique d'envisager et de comprendre les objets.

Identité

L'identité est la caractéristique qui permet de distinguer un objet parmi tous les autres au sein de la même classe. Par exemple, imaginez que deux voisins possèdent une voiture de la même marque, du même modèle et de la même couleur. Malgré la ressemblance évidente des véhicules, les plaques d'immatriculation seront uniques, prouvant l'identité propre de chaque voiture. La loi prévoit que chaque objet voiture doit être distinct.

(Comment fonctionneraient les assurances sans identification des voitures ?)

Comportement

Le comportement est la caractéristique qui rend les objets utiles. Les objets existent pour produire un comportement. La plupart du temps, vous ignorez les mécanismes internes de fonctionnement de la voiture et ne pensez qu'à son comportement essentiel. Les voitures sont utiles, car vous pouvez les conduire. Les mécanismes existent mais sont inaccessibles pour l'essentiel. C'est le comportement de l'objet qui est accessible. Le comportement d'un objet est également une caractéristique majeure pour déterminer sa classification. Les objets de la même classe partagent le même comportement. Une voiture est une voiture car vous pouvez la conduire, un stylo est un stylo car il vous permet d'écrire.

État

L'*état* fait référence aux mécanismes de fonctionnement internes d'un objet qui lui permettent de produire le comportement qui le définit. Un objet bien conçu conserve son état inaccessible. Cela est étroitement lié aux concepts d'abstraction et d'encapsulation. Vous ne voulez pas savoir comment un objet fait ce qu'il fait, vous voulez juste qu'il le fasse. La coïncidence peut faire que deux objets contiennent le même état, mais ils n'en restent pas moins deux objets différents. Par exemple, deux vrais jumeaux peuvent présenter exactement le même état (leur ADN) alors qu'ils sont deux individus distincts.

Comparaison entre classes et structs

■ Un struct est un modèle de valeur

- Pas d'identité, état accessible, pas de comportement supplémentaire

■ Une classe est un modèle d'objet

- Identité, état inaccessible, comportement supplémentaire

```
struct Time                class BankAccount
{
    public int hour;        {
    public int minute;      ...
}                          ...
}
```

Structs

Un struct, comme *Time* dans le code précédent, n'a pas d'identité. S'il existe deux variables *Time* représentant toutes deux l'heure 12:30, le programme se comportera exactement de la même manière, quelle que soit celle que vous utilisez. Les entités logicielles sans identité s'appellent des *valeurs*. Les types intégrés décrits dans le Module 3, « Utilisation des variables de type valeur », du cours 2132A, *Programmation en C#*, comme **int**, **bool**, **decimal**, et tous les types **struct**, s'appellent des *types valeur* en C#.

Les variables de type struct peuvent contenir des méthodes, mais cela n'est pas recommandé. Il est préférable qu'elles ne contiennent que des données. Toutefois, il est très raisonnable de définir des opérateurs dans les structs. Les opérateurs sont des méthodes stylisées qui n'ajoutent pas de nouveaux comportements, mais fournissent une syntaxe plus concise pour les comportements existants.

Classes

Une classe, comme **BankAccount** dans le code précédent, a une identité. S'il existe deux objets **BankAccount**, le programme se comportera différemment suivant celui que vous utilisez. Les entités logicielles qui possèdent une identité s'appellent des *objets*. (Les variables de type struct sont également parfois appelées objets, mais il s'agit à proprement parler de *valeurs*.) Les types représentés par des classes sont appelés des *types référence* en C#. Par rapport aux structs, rien à l'exception des méthodes ne doit être visible dans une classe bien conçue. Ces méthodes ajoutent un comportement de niveau supérieur en plus du comportement primitif présent dans les données inaccessibles des niveaux inférieurs.

Types valeur et types référence

Les types valeur sont les types présents au plus bas niveau d'un programme. Ce sont des éléments qui servent à construire des entités logicielles plus grandes. Les instances de types valeur peuvent être librement copiées et existent dans la pile en tant que variables locales ou en tant qu'attributs à l'intérieur des objets qu'elles décrivent.

Les types référence sont les types présents aux niveaux les plus élevés d'un programme. Ils sont élaborés à partir d'entités logicielles plus petites. Les instances de type référence ne peuvent généralement pas être copiées et existent sur le tas.

Abstraction

■ L'abstraction est une ignorance sélective

- Décidez ce qui est important et ce qui ne l'est pas
- Concentrez-vous sur ce qui est important et agissez en fonction
- Ignorez et ne dépendez pas de ce qui est sans importance
- Utilisez l'encapsulation pour mettre en œuvre l'abstraction

L'objectif de l'abstraction n'est pas d'être vague,
mais de créer un nouveau niveau sémantique dans lequel
il est possible d'être très précis.
Edsger Dijkstra

L'*abstraction* consiste à débarrasser une idée ou un objet de tous ses éléments superflus pour n'en conserver que la forme minimale et essentielle. Une bonne abstraction élimine les détails inutiles et vous permet de vous concentrer sur les éléments importants.

L'abstraction est un principe logiciel important. Une classe bien conçue expose un minimum de méthodes soigneusement sélectionnées qui assurent le comportement essentiel de la classe d'une façon simple à utiliser. Malheureusement, créer de bonnes abstractions logicielles n'est pas facile. Trouver de bonnes abstractions requiert généralement une connaissance complète du problème et de son contexte, une grande clarté de réflexion et beaucoup d'expérience.

Dépendance minimale

Les meilleures abstractions logicielles simplifient les choses complexes. Elles y parviennent en n'hésitant pas à masquer les aspects non essentiels d'une classe. Une fois réellement masqués, on ne peut plus les voir, les utiliser ou en dépendre de quelque manière que ce soit.

C'est ce principe de dépendance minimale qui rend l'abstraction si importante. La certitude de devoir un jour modifier le code est l'une des rares qui existent en matière de développement de logiciel. La compréhension totale, si elle arrive un jour, ne survient qu'à la fin du processus de développement. Les premières décisions sont prises à la lumière d'une compréhension partielle du problème et sont progressivement rectifiées. Les spécifications changent également une fois le problème mieux défini. Les versions futures s'enrichiront également de nouvelles fonctionnalités. Le changement est normal dans le développement de logiciel. Au mieux, vous pouvez en réduire l'impact. Et moins vous dépendez de quelque chose, moins vous êtes affecté lorsque cette chose change.

Exemples connexes

Pour illustrer le principe de dépendance minimale qui fait de l'abstraction un facteur si important, voici quelques exemples liés à cette question :

À mesure qu'elle s'approche de la perfection, la machine disparaît progressivement derrière sa fonction. Il semble que la perfection soit atteinte non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer. Au sommet de son évolution, la machine s'est totalement effacée.

—Antoine de Saint-Exupéry, *Du vent, du sable et des étoiles*

Le minimum peut se définir comme la perfection atteinte par un objet fabriqué lorsqu'il n'est plus possible de l'améliorer par soustraction. C'est la qualité que possède un objet lorsque chaque composant, chaque détail et chaque croisement a été réduit ou condensé à sa forme essentielle. C'est ce qui résulte de l'omission des superflus.

—John Pawson, *Minimum*

L'objectif principal de la communication est la clarté et la simplicité. Simplicité est synonyme d'effort canalisé.

—Edward de Bono, *Simplicité*

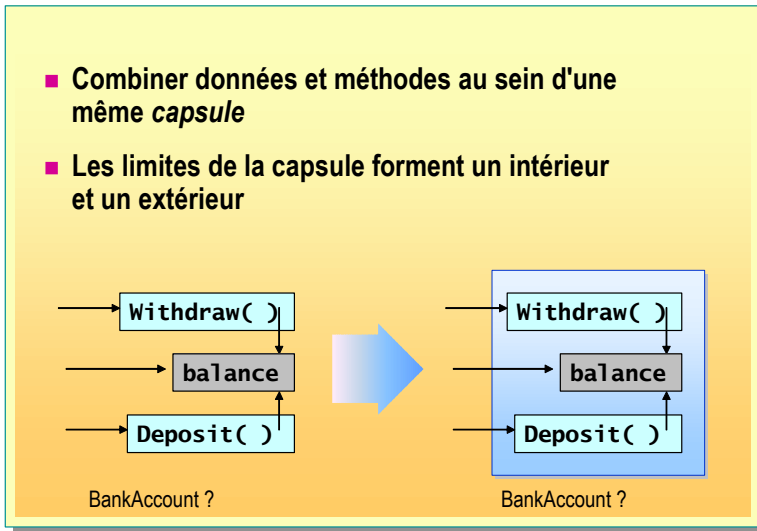
◆ Utilisation de l'encapsulation

- Combinaison de données et de méthodes
- Contrôle de la visibilité d'accès
- Pourquoi encapsuler ?
- Données d'objet
- Utilisation de données statiques
- Utilisation de méthodes statiques

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- combiner données et méthodes au sein d'une même capsule ;
- utiliser l'encapsulation au sein d'une classe ;
- utiliser des méthodes de données statiques dans une classe.

Combinaison de données et de méthodes



L'encapsulation revêt deux aspects importants :

- combiner les données et les fonctions dans une seule entité (traité dans la diapositive) ;
- contrôler l'accessibilité des membres de l'entité (traité dans la diapositive suivante).

Programmation procédurale

Les programmes procéduraux traditionnels écrits dans des langages tels que C contiennent essentiellement de grandes quantités de données et de nombreuses fonctions. Chaque fonction peut accéder à chaque élément de donnée. Cette approche couplée peut fonctionner dans un petit programme, mais elle devient rapidement ingérable à mesure que le programme s'étoffe. Changer le mode de représentation des données est synonyme de chaos. Toutes les fonctions qui font usage (et par conséquent dépendent) des données modifiées échouent. À mesure que le programme grossit, il est de plus en plus difficile d'y apporter la moindre modification. Le programme devient plus fragile et moins stable. L'approche fonction-données séparées n'est pas évolutive. Elle ne facilite pas le changement qui, comme le savent les développeurs, est la seule constante.

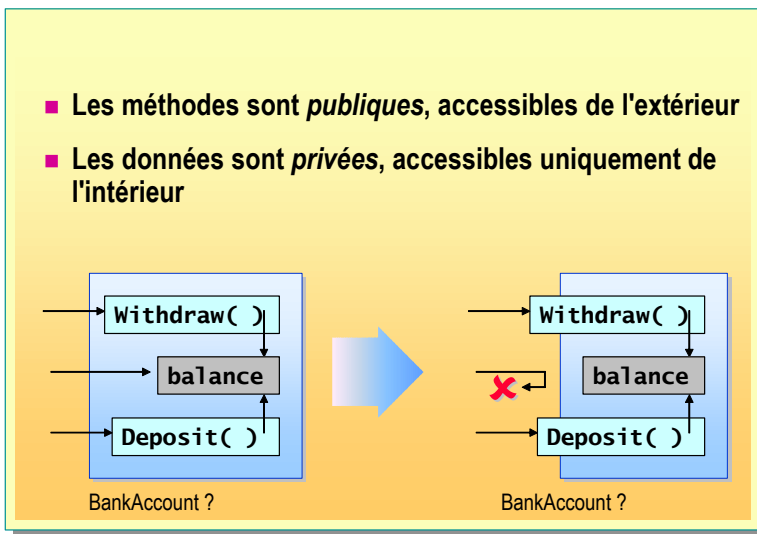
Séparer les données des fonctions pose aussi un autre problème. Cette technique n'est pas adaptée au raisonnement naturel de l'homme, en termes d'abstractions comportementales de haut niveau. Étant donné que des personnes (les programmeurs) écrivent les programmes, mieux vaut utiliser un modèle de programmation qui s'inspire du mode de réflexion humain plutôt que de la manière dont les ordinateurs sont actuellement construits.

Programmation orientée objet

La programmation orientée objet est apparue pour répondre à ces problèmes. Comprise et utilisée à bon escient, il s'agit en réalité d'une programmation orientée personne, car les personnes pensent et travaillent naturellement en fonction du comportement de haut niveau des objets.

Le premier pas, qui est aussi le plus important, vers la programmation orientée objet (et qui s'éloigne de la programmation procédurale) consiste à combiner les données et les fonctions au sein d'une entité.

Contrôle de la visibilité d'accès



Dans le graphique à gauche, **Withdraw** (débit), **Deposit** (crédit) et **balance** (solde) ont été groupés à l'intérieur d'une « capsule ». La diapositive suggère que le nom de la capsule est **BankAccount**. Toutefois, ce modèle de compte bancaire est erroné : les données de **balance** sont accessibles. (Imaginez que les vrais soldes bancaires soient ainsi accessibles : vous pourriez les augmenter sans faire de dépôts !) Les comptes bancaires ne fonctionnent pas ainsi : le problème et son modèle ont peu en commun.

Vous pouvez résoudre ce problème par l'encapsulation. Une fois les données et les fonctions combinées dans une seule entité, l'entité elle-même forme un ensemble fermé, qui crée naturellement un intérieur et un extérieur. Vous pouvez utiliser cet ensemble pour contrôler de manière sélective l'accès aux entités : certaines ne seront accessibles que de l'intérieur, d'autres de l'intérieur et de l'extérieur. Les membres qui sont toujours accessibles sont *publics* (public); ceux qui ne sont accessibles que de l'intérieur sont *privés* (private).

Pour rendre le modèle d'un compte bancaire plus proche de la réalité, vous pouvez rendre les méthodes **Withdraw** et **Deposit** publiques, et le solde **balance** privé. À présent, la seule manière d'augmenter le solde du compte bancaire de l'extérieur est de créditer le compte. Notez que **Deposit** peut accéder à **balance** car **Deposit** est à l'intérieur.

C#, comme de nombreux autres langages de programmation orientés objet, vous offre une liberté absolue en matière d'accessibilité des membres. Vous pouvez, si vous le souhaitez, créer des données publiques. Toutefois, il est recommandé que les données soient toujours marquées comme privées. (Certains langages de programmation imposent cette consigne.)

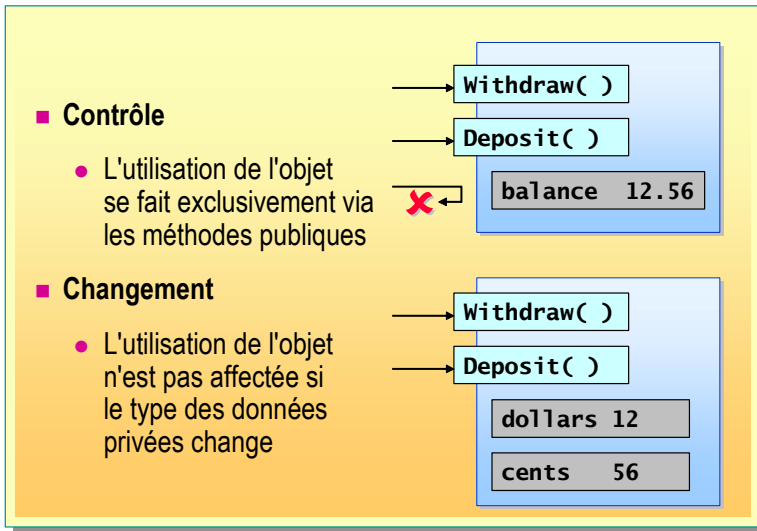
Les types dont la représentation des données est totalement privée s'appellent des types de données abstraits (ADT, *Abstract Data Types*). Ils sont abstraits dans la mesure où vous ne pouvez pas accéder à (et compter sur) la représentation privée des données. Vous pouvez utiliser uniquement les méthodes comportementales.

Les types intégrés tels que **int** sont, à leur manière, des ADT. Pour additionner deux variables d'entiers, vous n'avez pas besoin de connaître la représentation binaire interne de chaque valeur d'entier. La seule chose que vous devez connaître est le nom de la méthode qui effectue l'addition : l'opérateur d'addition (+).

Pour rendre des membres accessibles (publics), vous pouvez créer différentes vues de la même entité. La vue de l'extérieur est un sous-ensemble de la vue de l'intérieur. La vue restreinte est étroitement apparentée à l'idée d'abstraction : réduire une idée à sa forme essentielle.

Une grande partie du travail de conception consiste à déterminer si vous devez placer une fonctionnalité à l'intérieur ou à l'extérieur. Plus vous pouvez placer de fonctionnalités à l'intérieur (tout en conservant les possibilités d'utilisation), mieux c'est.

Pourquoi encapsuler ?



Les deux raisons d'encapsuler sont les suivantes :

- contrôler l'utilisation ;
- réduire l'impact des changements.

L'encapsulation permet le contrôle

La première raison pour encapsuler est le contrôle de l'utilisation. Lorsque vous conduisez une voiture, vous ne pensez qu'à la conduite, pas aux mécanismes internes de la voiture. Lorsque vous retirez de l'argent d'un compte, vous ne pensez pas à la manière avec laquelle le compte est représenté. Vous pouvez vous servir de l'encapsulation et des méthodes comportementales pour concevoir des objets logiciels qui peuvent être utilisés uniquement comme vous l'avez souhaité.

L'encapsulation permet le changement

La deuxième raison d'encapsuler dérive de la première. Si les détails d'implémentation d'un objet sont privés, les modifications apportées à ces détails n'affecteront pas directement les utilisateurs de l'objet (qui n'ont accès qu'aux méthodes publiques). En pratique, cela peut être extrêmement utile. Le nom des méthodes se stabilise généralement bien avant l'implémentation des méthodes.

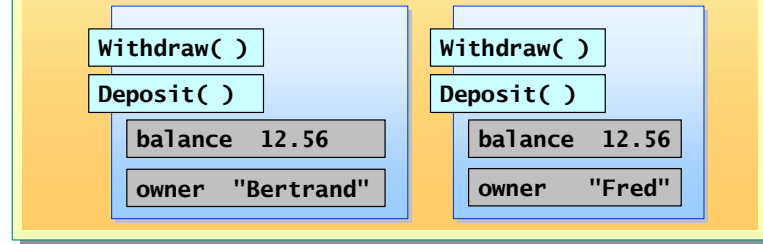
La capacité à effectuer des modifications internes est étroitement liée à l'idée d'abstraction. Face à deux conceptions possibles pour une classe, choisissez celle qui contient le moins de méthodes publiques.

En d'autres termes, si vous avez à choisir entre rendre une méthode publique ou privée, rendez-la privée. Une méthode privée peut être modifiée en toute liberté, voire rendue publique par la suite. En revanche, une méthode publique ne peut pas redevenir privée sans rupture du code client.

Données d'objet

■ **Les données d'objet décrivent les informations relatives aux objets *individuels***

- Par exemple, chaque compte bancaire possède son propre solde. Si deux comptes ont le même solde, ce n'est qu'une coïncidence.



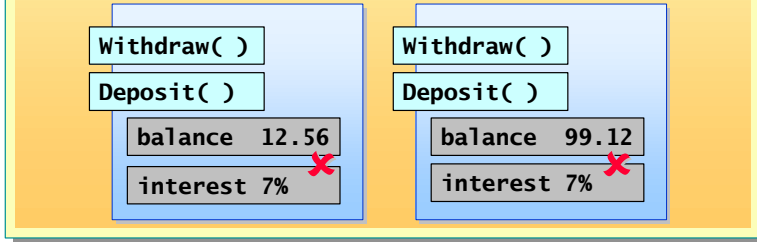
La plupart des données à l'intérieur d'un objet fournissent des informations sur cet objet particulier. Par exemple, chaque compte bancaire possède son propre solde. Il est bien sûr possible que plusieurs comptes bancaires présentent le même solde. Il s'agit dans ce cas d'une simple coïncidence.

Les données contenues dans un objet sont privées et ne sont accessibles qu'aux méthodes de l'objet. Cette encapsulation et séparation signifient qu'un objet est autonome.

Utilisation de données statiques

- **Les données statiques décrivent les informations relatives à tous les objets d'une classe**

- Par exemple, supposons que tous les comptes partagent le même taux d'intérêt. Stocker le taux d'intérêt dans chaque compte serait une mauvaise idée. Pourquoi ?



Il n'est pas toujours bienvenu de stocker des informations à l'intérieur de chaque objet. Par exemple, si tous les comptes bancaires partagent le même taux d'intérêt, le stockage du taux dans chaque objet compte est une mauvaise idée pour les raisons suivantes :

- C'est une mauvaise implémentation du problème décrit : « Tous les comptes bancaires partagent le même taux d'intérêt. »
- Il augmente inutilement la taille de chaque objet, et utilise des ressources mémoire supplémentaires lors de l'exécution du programme et de l'espace disque supplémentaire lors de l'enregistrement.
- Il rend difficile de changer le taux d'intérêt. Vous êtes obligé de changer le taux d'intérêt dans chaque objet compte. Si vous devez modifier le taux d'intérêt individuellement dans chaque objet, vous rendez les comptes inaccessibles le temps de la modification.
- Il augmente la taille de la classe. Les données de taux d'intérêt privées nécessiteraient des méthodes publiques. La classe de compte commence à perdre sa cohésion. Elle ne fait plus une seule chose, en la faisant bien.

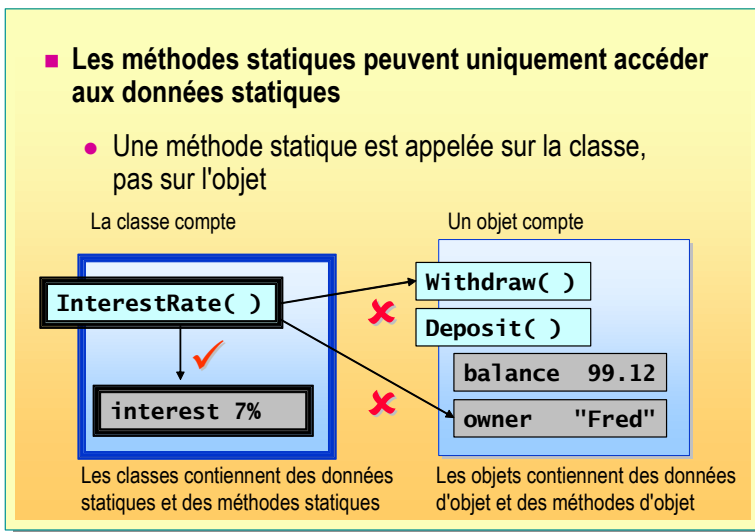
Pour résoudre ce problème, ne partagez pas d'informations communes aux différents objets au niveau des objets. Au lieu de décrire plusieurs fois le taux d'intérêt au niveau de l'objet, décrivez le taux d'intérêt une fois au niveau de la classe. Lorsque vous définissez le taux d'intérêt au niveau de la classe, il devient dans les faits une donnée globale.

Toutefois, les données globales, par définition, ne sont pas stockées dans une classe et ne peuvent donc pas être encapsulées. Pour cette raison, de nombreux langages de programmation orientés objet (tels que C#) n'autorisent pas l'utilisation de données globales. Ils autorisent à la place la description des données comme statiques.

Déclaration de données statiques

Les données statiques sont physiquement déclarées à l'intérieur d'une classe (qui est une entité de compilation statique) et bénéficient de l'encapsulation assurée par la classe, mais sont logiquement associées avec la classe elle-même, et non avec chaque objet. En d'autres termes, les données statiques sont déclarées à l'intérieur d'une classe par commodité syntaxique et existent même si le programme ne crée jamais d'objets de cette classe.

Utilisation de méthodes statiques



Vous utilisez des méthodes statiques pour encapsuler des données statiques. Dans l'exemple de la diapositive, le taux d'intérêt appartient à la classe account et pas à un objet account individuel. Il semble donc justifié de fournir au niveau de la classe des méthodes pouvant être utilisées pour accéder au taux d'intérêt ou le modifier.

Vous pouvez déclarer des méthodes comme étant statiques de la même manière que pour les données. Les méthodes statiques existent au niveau de la classe. Vous pouvez contrôler l'accessibilité des méthodes et des données statiques en utilisant des modificateurs d'accès, tels que public et private. En choisissant des méthodes statiques publiques et des données statiques privées, vous pouvez encapsuler des données statiques de la même manière que vous pouvez encapsuler des données d'objet.

Une méthode statique existe au niveau de la classe et est appelée par rapport à la classe, et non à un objet. Cela signifie qu'une méthode statique ne peut pas utiliser **this**, l'opérateur qui fait implicitement référence à l'objet effectuant un appel de méthode d'objet. En d'autres termes, une méthode statique ne peut pas accéder à des données ou à des méthodes non statiques. Les seuls membres d'une classe auxquels une méthode statique peut accéder sont les données statiques et les autres méthodes statiques.

Les méthodes statiques conservent l'accès à tous les membres privés d'une classe et peuvent accéder aux données non statiques privées par le biais d'une référence d'objet. Le code suivant fournit un exemple :

```
class Time
{
    ...
    public static void Reset(Time t)
    {
        t.hour = 0;    // Ok
        t.minute = 0;  // Ok
        hour = 0;      // erreur de compilation
        minute = 0;    // erreur de compilation
    }
    private int hour, minute;
}
```

◆ C# et orientation objet

- Hello, World revu et corrigé
- Définition de classes simples
- Instanciation de nouveaux objets
- Utilisation du mot clé this
- Création de classes imbriquées
- Accès aux classes imbriquées

Dans cette leçon, vous allez réexaminer le programme Hello, World d'origine. La structure du programme sera expliquée dans une perspective orientée objet.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- utiliser les mécanismes qui permettent à un objet d'en créer un autre en C# ;
- définir des classes imbriquées.

Hello, World revu et corrigé

```
using System;

class Hello
{
    public static int Main( )
    {
        Console.WriteLine("Hello, World");
        return 0;
    }
}
```

Le code de Hello, World est présenté sur la diapositive. On peut poser quelques questions et y répondre :

- Comment le runtime invoque-t-il les classes ?
- Pourquoi la méthode **Main** est-elle statique ?

Comment le runtime invoque-t-il les classes ?

S'il existe une seule méthode **Main**, le compilateur en fera automatiquement le point d'entrée du programme. Le code suivant fournit un exemple :

```
// OneEntrance.cs
class OneEntrance
{
    static void Main( )
    {
        ...
    }
}
// fin du fichier
```

```
c:\> csc OneEntrance.cs
```

Avertissement Le point d'entrée d'un programme C# doit être **Main** avec un « M » majuscule. La signature de **Main** est également importante.

Toutefois, s'il existe plusieurs méthodes appelées **Main**, l'une d'elles doit être explicitement désignée comme point d'entrée du programme (et cette méthode **Main** doit également être explicitement publique). Le code suivant fournit un exemple :

```
// TwoEntries.cs
using System;
class EntranceOne
{
    public static void Main()
    {
        Console.WriteLine("EntranceOne.Main( )");
    }
}
class EntranceTwo
{
    public static void Main()
    {
        Console.WriteLine("EntranceTwo.Main( )");
    }
}
// Fin du fichier
```

```
c:\> csc /main:EntranceOne TwoEntries.cs
c:\> twoentries.exe
EntranceOne.Main( )
c:\> csc /main:EntranceTwo TwoEntries.cs
c:\> twoentries.exe
EntranceTwo.Main( )
c:\>
```

Notez que l'option de ligne de commande respecte la casse. Si le nom de la classe contenant **Main** est **EntranceOne** (avec un E et un O en majuscule), la commande suivante ne fonctionnera pas :

```
c:\> csc /main:entranceone TwoEntries.cs
```

S'il n'existe pas de méthode **Main** dans le projet, vous ne pouvez pas créer de programme exécutable. Toutefois, vous pouvez créer une bibliothèque de liaisons dynamiques (DLL), comme suit :

```
// NoEntrance.cs
using System;
class NoEntrance
{
    public static void NotMain( )
    {
        Console.WriteLine("NoEntrance.NotMain( )");
    }
}
// Fin du fichier

c:\> csc /target:library NoEntrance.cs
c:\> dir
...
NoEntrance.dll
...
```

Pourquoi la méthode Main est-elle statique ?

Elle peut ainsi être invoquée sans que le runtime doive créer une instance de la classe.

Les méthodes non statiques peuvent uniquement être appelées sur un objet, comme l'illustre le code suivant :

```
class Example
{
    void NonStatic( ) { ... }
    static void Main( )
    {
        Example eg = new Example( );
        eg.NonStatic( ); // Se compile
        NonStatic( );    // erreur de compilation
    }
    ...
}
```

Si la méthode **Main** n'est pas statique, comme dans le code suivant, le runtime doit créer un objet afin de l'appeler.

```
class Example
{
    void Main( )
    {
        ...
    }
}
```

En d'autres termes, le runtime devrait effectivement exécuter le code suivant :

```
Example run = new Example( );
run.Main( );
```

Définition de classes simples

- Les données et les méthodes sont regroupées dans une classe
- Les méthodes sont publiques, les données privées

```
class BankAccount
{
    {
        public void Withdraw(decimal amount))
        { ... }
        public void Deposit(decimal amount)
        { ... }
        private decimal balance;
        private string name;
    }
}
```

Les méthodes
publiques
Décrivent le
comportement
accessible

Les champs
privés
Décrivent l'état
inaccessible

Bien que les classes et les structs soient sémantiquement différents, ils n'en possèdent pas moins quelques similarités syntaxiques. Pour définir une classe plutôt qu'un struct :

- Utilisez le mot clé **class** au lieu de **struct**.
- Déclarez vos données à l'intérieur de la classe exactement comme vous le feriez pour un struct.
- Déclarez vos méthodes à l'intérieur de la classe.
- Ajoutez des modificateurs d'accès aux déclarations de vos données et méthodes. Les deux modificateurs d'accès les plus simples sont **public** et **private**. (Les trois autres seront traités plus loin dans ce cours.)

Remarque C'est à vous de décider d'utiliser les modificateurs d'accès **public** et **private** à bon escient pour mettre en œuvre l'encapsulation. C# ne vous empêche pas de créer des données publiques.

La signification de *public* est « accès non limité ». La signification de *private* est « accès limité au type contenant ». L'exemple suivant explicite ce propos :

```
class BankAccount
{
    public void Deposit(decimal amount)
    {
        balance += amount;
    }
    private decimal balance;
}
```

Dans cet exemple, la méthode **Deposit** peut accéder au **balance** privé car **Deposit** est une méthode de **BankAccount** (le type qui contient **balance**). En d'autres termes, **Deposit** est à l'intérieur. De l'extérieur, les membres privés sont toujours inaccessibles. Dans l'exemple suivant, la compilation de l'expression `underAttack.balance` échouera.

```
class BankRobber
{
    public void StealFrom(BankAccount underAttack)
    {
        underAttack.balance -= 999999M;
    }
}
```

L'expression `underAttack.balance` ne sera pas compilée car elle se trouve à l'intérieur de la méthode **StealFrom** de la classe **BankRobber**. Seules les méthodes de la classe **BankAccount** peuvent accéder aux membres privés des objets **BankAccount**.

Pour déclarer des données statiques, suivez le schéma utilisé par les méthodes statiques (telles que **Main**), et faites précéder la déclaration de champ par le mot clé **static**. Le code suivant fournit un exemple :

```
class BankAccount
{
    public void Deposit(decimal amount) { ... }
    public static void Main( ) { ... }
    ...
    private decimal balance;
    private static decimal interestRate;
}
```


Si vous ne spécifiez pas de modificateur d'accès lorsque vous déclarez un membre de classe, celui-ci sera privé par défaut. En d'autres termes, les deux méthodes suivantes sont sémantiquement identiques :

```
class BankAccount
{
    ...
    decimal balance;
}

class BankAccount
{
    ...
    private decimal balance;
}
```

Conseils Bien que cela ne soit pas indispensable, il est recommandé d'écrire explicitement **private**.

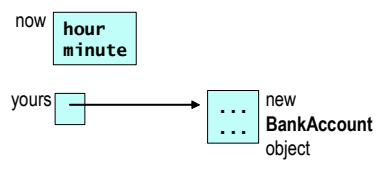
L'ordre de déclaration des membres d'une classe n'est pas important pour le compilateur C#. Toutefois, il est recommandé de déclarer les membres publics (méthodes) avant les membres privés (données). L'utilisateur d'une classe ne peut de toutes façons accéder qu'aux membres publics, et la déclaration des membres publics avant les membres privés reflète naturellement cette priorité.

Instanciation de nouveaux objets

- La déclaration d'une variable de classe n'entraîne pas la création d'un objet

- Utilisez l'opérateur **new** pour créer un objet

```
class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;
        BankAccount yours = new BankAccount( );
        yours.Deposit(999999M);
    }
}
```



Examinez les exemples de code suivants :

```
struct Time
{
    public int hour, minute;
}
class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;
        now.minute = 59;
        ...
    }
}
```

Les variables de type struct sont des *types valeur*. Cela signifie que lorsque vous déclarez une variable struct (comme *now* dans **Main**), vous créez une valeur dans la pile. Dans cet exemple, le struct *Time* contient deux entiers, de sorte que la déclaration de *now* crée deux entiers dans la pile, *now.hour* et *now.minute*. Ces deux entiers ne sont PAS initialisés par défaut à zéro. Par conséquent, la valeur de *now.hour* ou *now.minute* ne peut pas être lue avant d'avoir reçu une valeur définie. Les valeurs ont une portée au niveau du bloc dans lequel elles sont déclarées. Dans cet exemple, la portée de *now* s'étend à **Main**. Cela signifie que lorsque le flux de contrôle quitte **Main** (suite à un retour normal ou à une exception), *now* quitte sa portée et cesse d'exister.

Les classes sont complètement différentes, comme le montre le code suivant :

```
class Time // REMARQUE : Time est désormais une classe
{
    public int hour, minute;
}
class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;
        now.minute = 59;
        ...
    }
}
```

Lorsque vous déclarez une variable de classe, vous ne créez pas une instance ou un objet de cette classe. Dans ce cas, la déclaration de *now* ne crée pas un objet de la classe **Time**. Déclarer une variable de classe crée une référence qui est capable de faire référence à un objet de cette classe. C'est pourquoi les classes sont appelées des *types référence*. Cela signifie que si le runtime était autorisé à exécuter le code précédent, il essaierait d'accéder aux entiers à l'intérieur d'un objet *Time* qui n'existe pas. Par chance, le compilateur vous avertirait de cette erreur. Si vous compilez le code précédent, vous obtenez le message d'erreur suivant :

erreur CS0165: Utilisation d'une variable locale non assignée
'now'

Pour corriger cette erreur, vous devez créer un objet *Time* (à l'aide du mot clé **new**) et faire en sorte que la variable de référence *now* fasse effectivement référence au nouvel objet créé, comme dans le code suivant :

```
class Program
{
    static void Main( )
    {
        Time now = new Time( );
        now.hour = 11;
        now.minute = 59;
        ...
    }
}
```

N'oubliez pas que lorsque vous créez une valeur de struct locale sur la pile, les champs ne sont PAS initialisés à zéro par défaut. Les classes sont différentes : lorsque vous créez un objet en tant qu'instance d'une classe, comme décrit précédemment, les champs de l'objet sont initialisés à zéro par défaut. Par conséquent, le code ci-dessous se compile correctement :

```
class Program
{
    static void Main( )
    {
        Time now = new Time( );
        Console.WriteLine(now.hour);    // écrit 0
        Console.WriteLine(now.minute); // écrit 0
        ...
    }
}
```

Utilisation du mot clé **this**

- Le mot clé **this** fait référence à l'objet utilisé pour appeler la méthode

- Utile lorsque des identificateurs de portées différentes sont en conflit

```
class BankAccount
{
    ...
    public void SetName(string name)
    {
        {
            this.name = name;; ← Si cette instruction était
                                name = name;
                                Que se passerait-il ?
        }
        private string name;
    }
}
```

Le mot clé **this** fait implicitement référence à l'objet qui effectue un appel de méthode d'objet.

Dans le code suivant, l'instruction `name = name` n'aurait aucun effet. La raison en est que l'identificateur *name* à gauche de l'instruction ne correspond pas au champ **BankAccount** privé appelé *name*. Les deux identificateurs correspondent au paramètre de méthode qui s'appelle également *name*.

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

Avertissement Le compilateur C# n'affiche PAS d'avertissement pour ce bogue.

Utilisation du mot clé **this**

Vous pouvez résoudre ce problème de référence à l'aide du mot clé **this**, comme le présente la diapositive. Le mot clé **this** fait référence à l'objet pour lequel la méthode est actuellement appelée.

Remarque Les méthodes statiques ne peuvent pas utiliser **this** car elles ne sont pas appelées par l'utilisation d'un objet.

Changement de nom du paramètre

Vous pouvez également résoudre le problème de référence en changeant le nom du paramètre, comme dans l'exemple suivant :

```
class BankAccount
{
    public void SetName(string newName)
    {
        name = newName;
    }
    private string name;
}
```

Conseil L'utilisation de **this** en écrivant des constructeurs est un idiome commun de C#. Le code suivant fournit un exemple :

```
struct Time
{
    public Time(int hour, int minute)
    {
        this.hour = hour;
        this.minute = minute;
    }
    private int hour, minute;
}
```

Conseil Le mot clé **this** est également utilisé pour implémenter un chaînage d'appels. Notez que dans la classe suivante les deux méthodes renvoient l'objet qui appelle :

```
class Book
{
    public Book SetAuthor(string author)
    {
        this.author = author;
        return this;
    }
    public Book SetTitle(string title)
    {
        this.title = title;
        return this;
    }
    private string author, title;
}
```

Renvoyer **this** permet aux appels de méthode d'être chaînés les uns aux autres, comme suit :

```
class Usage
{
    static void Chained(Book good)
    {
        good.SetAuthor("Fowler").SetTitle("Refactoring");
    }
    static void NotChained(Book good)
    {
        good.SetAuthor("Fowler");
        good.SetTitle("Refactoring");
    }
}
```

Remarque Une méthode statique existe au niveau de la classe et est appelée par rapport à la classe, pas par rapport à un objet. Cela signifie qu'une méthode statique ne peut pas utiliser l'opérateur **this**.

Création de classes imbriquées

- Les classes peuvent être imbriquées dans d'autres classes

```
class Program
{
    static void Main( )
    {
        Bank.Account yours = new Bank.Account( );
    }
}
class Bank
{
    ... class Account { .... }
}
```

Le nom complet de la classe imbriquée inclut le nom de la classe extérieure

Il existe cinq sortes de types différents dans C#:

- class
- struct
- interface
- enum
- delegate

Vous pouvez tous les imbriquer à l'intérieur d'une classe ou d'une structure.

Remarque Vous ne pouvez pas imbriquer de type à l'intérieur d'une interface, d'un enum ou d'un delegate.

Dans le code ci-dessus, la classe **Account** est imbriquée à l'intérieur de la classe **Bank**. Le nom complet de la classe imbriquée est **Account.Bank**, et c'est ce nom qui doit être employé pour nommer le type imbriqué en dehors de la portée de la classe **Bank**. Le code suivant fournit un exemple :

```
// Program.cs
class Program
{
    static void Main( )
    {
        Account yours = new Account( ); // erreur de
                                         // compilation
    }
}
// fin de fichier
c:\> csc Program.cs
error CS0246: Le type...'Account' est introuvable...
```


Par contre, le simple nom **Account** suffit à l'intérieur de **Bank**, comme dans l'exemple suivant :

```
class Bank
{
    class Account { ... }

    Account OpenAccount( )
    {
        return new Account( );
    }
}
```

Remarque Consultez la rubrique suivante pour un examen plus détaillé de l'exemple.

Les classes imbriquées offrent plusieurs fonctionnalités utiles :

- Les classes imbriquées peuvent être déclarées avec des modes d'accès spécifiques. Cet aspect est traité dans la rubrique suivante.
- L'utilisation de classes imbriquées supprime des noms supplémentaires de la portée globale ou de l'espace de noms contenant.
- Les classes imbriquées permettent à des structures supplémentaires d'être exprimées dans la grammaire du langage. Par exemple, le nom de la classe est **Bank.Account** (trois éléments) plutôt que **BankAccount** (un élément).

Accès aux classes imbriquées

- Les classes imbriquées peuvent aussi être déclarées publiques ou privées

```
class Bank
{
    public class Account { .... }
    private class AccountNumberGenerator { .... }
}
class Program
{
    static void Main( )
    {
        Bank.Account                accessible; ✓
        Bank.AccountNumberGenerator inaccessible; ✗
    }
}
```

Vous contrôlez l'accessibilité des données et des méthodes en les déclarant publiques ou privées. Vous contrôlez l'accessibilité d'une classe imbriquée exactement de la même manière.

Classe imbriquée publique

Une classe imbriquée publique n'impose aucune restriction d'accès. Elle est déclarée comme étant d'accès public. Le nom complet d'une classe imbriquée doit être employé à l'extérieur de la classe qui la contient.

Classe imbriquée privée

Une classe imbriquée privée dispose exactement des mêmes restrictions d'accès que les données ou méthodes privées. Une classe imbriquée privée est inaccessible de l'extérieur de la classe qui la contient, comme le montre l'exemple suivant :

```
class Bank
{
    private class AccountNumberGenerator
    {
        ...
    }
}
class Program
{
    static void Main( )
    {
        // erreur de compilation
        Bank.AccountNumberGenerator variable;
    }
}
```

Dans cet exemple, **Main** ne peut pas utiliser **Bank.AccountNumberGenerator** car **Main** est une méthode de **Program**, **AccountNumberGenerator** est privé et donc inaccessible à sa classe extérieure, **Bank**.

Une classe imbriquée privée est accessible aux seuls membres de la classe qui la contient, comme le montrent les exemples suivants :

```
class Bank
{
    public class Account
    {
        public void Setup( )
        {
            NumberSetter.Set(this);
            balance = 0M;
        }

        private class NumberSetter
        {
            public static void Set(Account a)
            {
                a.number = nextNumber++;
            }
            private static int nextNumber = 2311;
        }

        private int number;
        private decimal balance;
    }
}
```

Dans ce code, notez que la méthode **Account.Setup** peut accéder à la classe **NumberSetter** car, bien que **NumberSetter** soit une classe privée, elle est privée pour **Account**, et **Setup** est une méthode de **Account**.

Notez également que la méthode **Account.NumberSetter.Set** peut accéder au champ privé *balance* de l'objet **Account a**. La raison à cela est que **Set** est une méthode de la classe **NumberSetter**, qui est imbriquée dans **Account**. Par conséquent, **NumberSetter** (et ses méthodes) ont accès aux membres privés de **Account**.

L'accessibilité par défaut d'une classe imbriquée est privée (comme dans le cas des données et des méthodes). Dans l'exemple suivant, la classe **Account** est privée par défaut :

```
class Bank
{
    class Account { ... }

    public Account OpenPublicAccount( )
    {
        Account opened = new Account( );
        opened.Setup( );
        return opened;
    }

    private Account OpenPrivateAccount( )
    {
        Account opened = new Account( );
        opened.Setup( );
        return opened;
    }
}
```

La classe **Account** est accessible aux méthodes **OpenPublicAccount** et **OpenPrivateAccount** car elles sont toutes deux imbriquées à l'intérieur de **Bank**. Toutefois, la méthode **OpenPublicAccount** ne se compilera pas. Le problème est que **OpenPublicAccount** est une méthode publique, utilisable comme dans le code suivant :

```
class Program
{
    static void Main( )
    {
        Bank b = new Bank( );
        Bank.Account opened = b.OpenPublicAccount( );
        ...
    }
}
```

Ce code ne se compilera car **Bank.Account** n'est pas accessible à **Program.Main**, **Bank.Account** est privé pour **Bank**, et **Main** n'est pas une méthode de **Bank**. Le message d'erreur suivant apparaît :

```
error CS0050: Accessibilité incohérente : le type de retour
'Bank.Account' est moins accessible que la méthode
'Bank.OpenPublicAccount()'
```

Les règles d'accessibilité d'une classe de haut niveau (c'est-à-dire une classe qui n'est pas imbriquée dans une autre) ne sont pas les mêmes que celles des classes imbriquées. Une classe de haut niveau ne peut pas être déclarée privée et fonctionne selon les principes de l'accessibilité interne par défaut. (L'accès interne est traité de manière exhaustive dans un module ultérieur.)

Atelier 7.1 : Création et utilisation de classes



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer des classes et instancier des objets ;
- utiliser des données et des méthodes non-statiques ;
- utiliser des données et des méthodes statiques.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les techniques suivantes :

- créer des méthodes en C# ;
- passer des arguments en tant que paramètres de méthode en C#.

Durée approximative de cet atelier : 45 minutes

Exercice 1

Création et utilisation d'une classe

Dans cet exercice, vous reprenez le struct de compte bancaire que vous avez développé dans un module précédent et le convertissez en classe. Vous allez déclarer ses membres de données comme privés mais fournir des méthodes publiques non statiques pour accéder aux données. Vous allez créer un test de validation qui crée un objet compte (account) et le renseigne avec un numéro de compte et un solde spécifié par l'utilisateur. Enfin, vous allez imprimer les données du compte.

► Pour transformer le struct **BankAccount** en classe

1. Ouvrez le projet CreateAccount.sln qui se trouve dans le dossier *dossier d'installation*\Labs\Lab07\Starter\CreateAccount.
2. Étudiez le code qui se trouve dans le fichier BankAccount.cs. Notez que *BankAccount* est de type struct.
3. Compilez et exécutez le programme. Vous serez invité à entrer un numéro de compte et un solde initial. Répétez cette procédure pour créer un autre compte.
4. Modifiez *BankAccount* dans BankAccount.cs pour en faire une classe plutôt qu'une structure.
5. Compilez le programme. La compilation échouera. Ouvrez le fichier CreateAccount.cs et consultez la classe **CreateAccount**. La classe ressemblera à ceci :

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created;
        ...
        created.accNo = number; // Une erreur ici
        ...
    }
    ...
}
```

6. L'assignation à *created.accNo* s'est compilée sans erreur lorsque **BankAccount** était un struct. À présent qu'il s'agit d'une classe, elle ne se compile plus. La raison en est que lorsque **BankAccount** était un struct, la déclaration de la variable *created* créait une *valeur BankAccount* (dans la pile). À présent que **BankAccount** est une classe, la déclaration de la variable *created* ne crée pas de valeur **BankAccount**, mais crée une *référence BankAccount* qui ne fait pas encore référence à un *objet BankAccount*.

7. Changez la déclaration de *created* de sorte qu'elle soit initialisée par un nouvel objet **BankAccount**, comme suit :

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created = new BankAccount( );
        ...
        created.accNo = number;
        ...
    }
    ...
}
```

8. Enregistrez votre travail
9. Compilez et exécutez le programme. Vérifiez que les données entrées dans la console sont correctement lues et affichées dans la méthode **CreateAccount.Write**.

► Pour encapsuler la classe **BankAccount**

1. Tous les membres de données de la classe **BankAccount** sont actuellement publics. Modifiez-les pour les rendre privés, comme suit :

```
class BankAccount
{
    private long accNo;
    private decimal accBal;
    private AccountType accType;
}
```

2. Compilez le programme. La compilation échouera. L'erreur se produit dans la classe **CreateAccount** comme suit :

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created = new BankAccount( );
        ...
        created.accNo = number; // Une erreur ici à nouveau
        ...
    }
    ...
}
```

- La compilation des assignations de membres de données **BankAccount** échoue désormais car les membres de données sont privés. Seules les méthodes **BankAccount** peuvent accéder aux membres de données **BankAccount** privés. Vous devez écrire une méthode **BankAccount** publique pour effectuer les assignation à votre place. Effectuez la procédure ci-dessous :

Ajoutez une méthode non statique appelée **Populate** dans **BankAccount**. Cette méthode renverra **void** et attendra deux paramètres : un long (le numéro du compte bancaire) et un décimal (le solde du compte bancaire). Le corps de cette méthode assignera le paramètre long au champ *accNo* et le paramètre décimal au champ *accBal*. Il définira également le champ *accType* à **AccountType.Checking** comme suit :

```
class BankAccount
{
    public void Populate(long number, decimal balance)
    {
        accNo = number;
        accBal = balance;
        accType = AccountType.Courant;
    }

    private long accNo;
    private decimal accBal;
    private AccountType accType;
}
```

- Mettez en commentaires les trois assignations à la variable *created* dans la méthode **CreateAccount.NewBankAccount**. À leur place, ajoutez une instruction qui appelle la méthode **Populate** sur la variable *created*, en passant **number** et **balance** en tant qu'arguments. Cette transformation ressemblera à ceci :

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created = new BankAccount( );
        ...
        // created.accNo = number;
        // created.accBal = balance;
        // created.accType = AccountType.Courant;

        created.Populate(number, balance);
        ...
    }
    ...
}
```

- Enregistrez votre travail.

6. Compilez le programme. La compilation échoue. Il reste trois instructions dans la méthode **CreateAccount.Write** qui essaient d'accéder directement aux champs **BankAccount** privés. Vous devez écrire trois méthodes **BankAccount** publiques qui renvoient les valeurs de ces trois champs. Effectuez les étapes suivantes :

- a. Ajoutez une méthode publique non statique appelée **Number** dans **BankAccount**. Cette méthode renverra un paramètre long et n'attendra pas de paramètres. Elle renverra la valeur du champ *accNo* comme suit :

```
class BankAccount
{
    public void Populate(...) ...

    public long Number( )
    {
        return accNo;
    }
    ...
}
```

- b. Ajoutez une méthode publique non statique appelée **Balance** dans **BankAccount**, comme dans le code suivant. Cette méthode renverra un décimal et n'attendra pas de paramètres. Elle renverra la valeur du champ *accBal*.

```
class BankAccount
{
    public void Populate(...) ...

    ...
    public decimal Balance( )
    {
        return accBal;
    }
    ...
}
```

- c. Ajoutez une méthode publique non statique appelée **Type** dans **BankAccount**, comme dans le code suivant. Cette méthode renverra un **AccountType** et n'attendra pas de paramètres. Elle renverra la valeur du champ *accType*.

```
class BankAccount
{
    public void Populate(...) ...

    ...
    public AccountType Type( )
    {
        return accType;
    }
    ...
}
```

- d. Enfin, remplacez les trois instructions de la méthode **CreateAccount.Write** qui essaient d'accéder directement aux champs **BankAccount** privés par des appels aux trois méthodes publiques que vous venez de créer, comme suit :

```
class CreateAccount
{
    ...
    static void Write(BankAccount toWrite)
    {
        Console.WriteLine("Le numéro du compte est {0}",
            toWrite.Number( ));
        Console.WriteLine("Le solde du compte est {0}",
            toWrite.Balance( ));
        Console.WriteLine("Le type du compte est {0}",
            toWrite.Type( ));
    }
}
```

7. Enregistrez votre travail
8. Compilez le programme et corrigez les erreurs éventuelles qui s'y trouvent. Exécutez le programme. Vérifiez que les données entrées sur la console et passées à la méthode **BankAccount.Populate** sont correctement lues et affichées dans la méthode **CreateAccount.Write**.

► Pour poursuivre l'encapsulation de la classe **BankAccount**

1. Changez la méthode **BankAccount.Type** de sorte qu'elle renvoie le type du compte en tant que chaîne plutôt que comme un enum **AccountType**, comme suit :

```
class BankAccount
{
    ...
    public string Type( )
    {
        return accType.ToString( );
    }
    ...
    private AccountType accType;
}
```

2. Enregistrez votre travail
3. Compilez le programme et corrigez les éventuelles erreurs qui s'y trouvent. Exécutez le programme. Vérifiez que les données entrées dans la console et passées à la méthode **BankAccount.Populate** sont correctement lues et affichées dans la méthode **CreateAccount.Write**.

Exercice 2

Génération de numéros de compte

Dans cet exercice, vous allez modifier la classe **BankAccount** de l'exercice 1 de sorte qu'elle ne génère que des numéros de compte uniques. Pour ce faire, vous utiliserez une variable statique dans la classe **BankAccount** et une méthode qui incrémente et renvoie la valeur de cette variable. Lorsque le test de validation créera un nouveau compte, il appellera cette méthode pour générer le numéro de compte. Il appellera ensuite la méthode de la classe **BankAccount** qui définit le numéro du compte, en passant cette valeur comme un paramètre.

► **Pour garantir que chaque numéro de BankAccount est unique**

1. Ouvrez le projet UniqueNumbers.sln qui se trouve dans le dossier *dossier d'installation\Labs\Lab07\Starter\UniqueNumbers*.

Remarque Ce projet est identique à celui du projet CreateAccount déjà réalisé au cours de l'exercice 1.

2. Ajoutez un paramètre long statique privé appelé *nextAccNo* dans la classe **BankAccount**, comme suit :

```
class BankAccount
{
    ...
    private long accNo;
    private decimal accBal;
    private AccountType accType;

    private static long nextAccNo;
}
```

3. Ajoutez une méthode statique publique appelée **NextNumber** dans la classe **BankAccount**, comme dans le code suivant. Cette méthode renverra un paramètre long et n'attendra pas de paramètres. Elle renverra la valeur du champ *nextAccNo* en plus de l'incrémenter.

```
class BankAccount
{
    ...
    public static long NextNumber( )
    {
        return nextAccNo++;
    }

    private long accNo;
    private decimal accBal;
    private AccountType accType;

    private static long nextAccNo;
}
```

4. Mettez en commentaire l'instruction de la méthode **CreateAccount.NewBankAccount** qui écrit une invite sur la console demandant le numéro du compte bancaire, comme suit :

```
//Console.WriteLine("Entrez le numéro du compte : ");
```

5. Remplacez l'initialisation de *number* dans la méthode **CreateAccount.NewBankAccount** par un appel à la méthode **BankAccount.NextNumber** que vous venez de créer, comme suit :

```
//long number = long.Parse(Console.ReadLine( ));
long number = BankAccount.NextNumber( );
```

6. Enregistrez votre travail
7. Compilez le programme et corrigez les éventuelles erreurs qui s'y trouvent. Exécutez le programme. Vérifiez que les deux comptes ont les numéros 0 et 1.
8. Le champ statique **BankAccount.nextAccNo** est actuellement initialisé à zéro par défaut. Initialisez-le explicitement à 123.
9. Compilez et exécutez le programme. Vérifiez que les deux comptes créés ont les numéros 123 et 124.

► Pour poursuivre l'encapsulation de la classe **BankAccount**

1. Changez la méthode **BankAccount.Populate** de sorte qu'elle n'attende qu'un seul paramètre : le *balance* décimal. À l'intérieur de la méthode, assignez le champ *accNo* à l'aide de la méthode statique **BankAccount.NextNumber**, comme suit :

```
class BankAccount
{
    public void Populate(decimal balance)
    {
        accNo = NextNumber( );
        accBal = balance;
        accType = AccountType.Courant;
    }
    ...
}
```

2. Transformez **BankAccount.NextNumber** en une méthode privée, comme suit :

```
class BankAccount
{
    ...
    private static long NextNumber( ) ...
}
```

3. Mettez en commentaire la déclaration et l'initialisation de *number* dans la méthode **CreateAccount.NewBankAccount**. Changez l'appel de méthode **created.Populate** de sorte qu'elle ne passe qu'un seul paramètre, comme suit :

```
class CreateAccount
{
    ...
    static BankAccount NewBankAccount( )
    {
        BankAccount created = new BankAccount( );

        //long number = BankAccount.NextNumber( );
        ...
        created.Populate(balance);
        ...
    }
    ...
}
```

4. Enregistrez votre travail
5. Compilez le programme et corrigez les éventuelles erreurs qui s'y trouvent. Exécutez le programme. Vérifiez que les deux comptes ont encore les numéros 123 et 124.

Exercice 3

Ajout de méthodes publiques supplémentaires

Dans cet exercice, vous allez ajouter deux méthodes dans la classe **Account** : **Withdraw** (retrait) et **Deposit** (dépot).

Withdraw prendra un paramètre décimal et déduira sa valeur du solde. Toutefois, elle vérifiera d'abord que le compte est suffisamment approvisionné, car les découverts ne sont pas autorisés. Elle renverra une valeur booléenne indiquant si le retrait a réussi.

Deposit utilisera également un paramètre décimal dont elle ajoutera la valeur au solde du compte. Elle renverra ensuite la nouvelle valeur du solde.

► Pour ajouter une méthode **Deposit** à la classe **BankAccount**

1. Ouvrez le projet `MoreMethods.sln` qui se trouve dans le dossier *dossier d'installation\Labs\Lab07\Starter\MoreMethods*.

Remarque Ce projet est identique à celui du projet `UniqueNumbers` déjà réalisé au cours de l'exercice 2.

2. Ajoutez une méthode non statique publique appelée **Deposit** dans la classe **BankAccount**, comme dans le code suivant. Cette méthode utilisera également un paramètre décimal dont elle ajoutera la valeur au solde du compte. Elle renverra ensuite la nouvelle valeur du solde.

```
class BankAccount
{
    ...
    public decimal Deposit(decimal amount)
    {
        accBal += amount;
        return accBal;
    }
    ...
}
```

3. Ajoutez une méthode statique publique appelée **TestDeposit** dans la classe **CreateAccount**, comme dans le code suivant. Cette méthode renverra **void** et attendra un paramètre **BankAccount**. La méthode écrira une invite sur la console demandant à l'utilisateur le montant du dépôt, capturera le montant entré comme décimal puis appellera la méthode **Deposit** sur le paramètre **BankAccount**, en passant le montant comme argument.

```
class CreateAccount
{
    ...
    public static void TestDeposit(BankAccount acc)
    {
        Console.WriteLine("Entrer le montant du dépôt : ");
        decimal amount = decimal.Parse(Console.ReadLine());
        acc.Deposit(amount);
    }
    ...
}
```

4. Ajoutez à **CreateAccount.Main** des instructions qui appellent la méthode **TestDeposit** que vous venez de créer, comme dans le code suivant. Veillez à appeler **TestDeposit** pour les deux objets compte. Utilisez la méthode **CreateAccount.Write** pour afficher le compte après que le dépôt ait été effectué.

```
class CreateAccount
{
    static void Main( )
    {
        BankAccount berts = NewBankAccount( );
        Write(berts);
        TestDeposit(berts);
        Write(berts);

        BankAccount freds = NewBankAccount( );
        Write(freds);
        TestDeposit(berts);
        Write(freds);
    }
}
```

5. Enregistrez votre travail
6. Compilez le programme et corrigez les éventuelles erreurs qui s'y trouvent. Exécutez le programme. Vérifiez que les dépôts fonctionnent comme prévu.

Remarque Si vous en avez le temps, vous pouvez également ajouter un contrôle à **Deposit** pour garantir que le paramètre décimal passé n'est pas négatif.

► **Pour ajouter une méthode `Withdraw` à la classe `BankAccount`**

1. Ajoutez une méthode non statique publique appelée **`Withdraw`** dans **`BankAccount`**, comme dans le code suivant. Cette méthode attendra un paramètre décimal spécifiant le montant du retrait. Elle ne déduira le montant du solde que si le compte est suffisamment approvisionné, car les découverts ne sont pas autorisés. Elle renverra une valeur booléenne indiquant si le retrait a réussi.

```
class BankAccount
{
    ...
    public bool Withdraw(decimal amount)
    {
        bool sufficientFunds = accBal >= amount;
        if (sufficientFunds) {
            accBal -= amount;
        }
        return sufficientFunds;
    }
    ...
}
```

2. Ajoutez une méthode statique publique appelée **`TestWithdraw`** dans la classe **`CreateAccount`**, comme dans le code suivant. Cette méthode renverra **`void`** et attendra un paramètre **`BankAccount`**. La méthode écrira une invite sur la console demandant à l'utilisateur le montant du retrait, capturera le montant entré comme décimal puis appellera la méthode **`Withdraw`** sur le paramètre **`BankAccount`**, en passant le montant comme argument. La méthode capturera le résultat booléen renvoyé par **`Withdraw`** et écrira un message sur la console en cas d'échec du retrait.

```
class CreateAccount
{
    ...
    public static void TestWithdraw(BankAccount acc)
    {
        Console.WriteLine("Entrer le montant du retrait : ");
        decimal amount = decimal.Parse(Console.ReadLine());
        if (!acc.Withdraw(amount)) {
            Console.WriteLine("Provision insuffisante.");
        }
    }
    ...
}
```


3. Ajoutez à **CreateAccount.Main** des instructions qui appellent la méthode **TestWithdraw** que vous venez de créer, comme dans le code suivant. Veillez à appeler **TestWithdraw** pour les deux objets compte. Utilisez la méthode **CreateAccount.Write** pour afficher le compte après que le retrait ait été effectué.

```
class CreateAccount
{
    static void Main( )
    {
        BankAccount berts = NewBankAccount( );
        Write(berts);
        TestDeposit(berts);
        Write(berts);
        TestWithdraw(berts);
        Write(berts);

        BankAccount freds = NewBankAccount( );
        Write(freds);
        TestDeposit(berts);
        Write(freds);
        TestWithdraw(freds);
        Write(freds);
    }
}
```

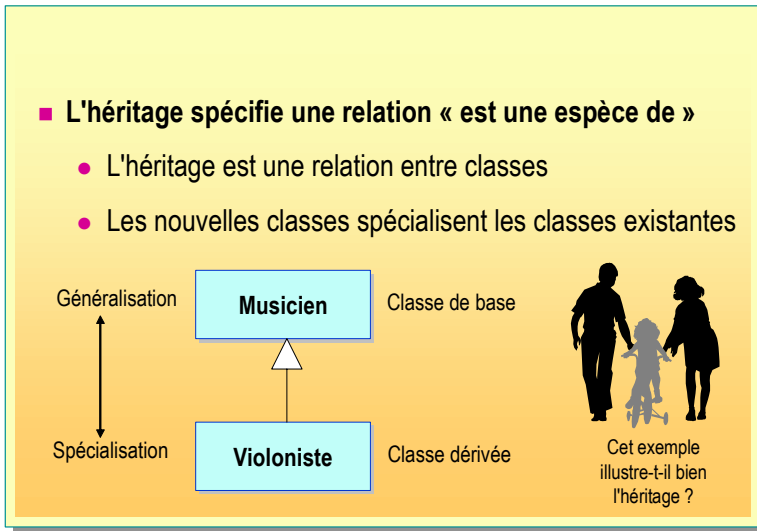
4. Enregistrez votre travail
5. Compilez le programme et corrigez les éventuelles erreurs qui s'y trouvent. Exécutez le programme. Vérifiez que les retraits fonctionnent comme prévu. Testez des retraits réussis et ratés.

◆ Définition de systèmes orientés objet

- Héritage
- Hiérarchies des classes
- Héritage simple et multiple
- Polymorphisme
- Classes de base abstraites
- Interfaces
- Liaisons anticipée et tardive

Dans cette leçon, nous allons aborder l'héritage et le polymorphisme. Vous apprendrez à implémenter ces concepts en C# dans les modules ultérieurs.

Héritage



L'héritage est une relation qui est spécifiée au niveau de la classe. Une nouvelle classe peut être dérivée d'une classe existante. Dans la diapositive ci-dessus, la classe **Violoniste** est dérivée de la classe **Musicien**. La classe **Musicien** s'appelle la classe de *base* (ou, moins fréquemment, la classe parente, ou super-classe). La classe **Violoniste** s'appelle la classe *dérivée* (ou, moins fréquemment, la classe fille, ou sous-classe). L'héritage est montré à l'aide de la notation UML (Unified Modeling Language). D'autres notations UML seront couvertes dans les diapositives ultérieures.

L'héritage est une relation puissante car une classe dérivée hérite de toutes les caractéristiques de sa classe de base. Par exemple, si la classe de base **Musicien** contient une méthode appelée **AccorderVotreInstrument**, cette méthode devient automatiquement membre de la classe dérivée **Violoniste**.

Une classe de base peut posséder un nombre illimité de classes dérivées. Par exemple, de nouvelles classes (comme **Guitariste** ou **Pianiste**) pourraient toutes dériver de la classe **Musicien**. Ces nouvelles classes dérivées hériteraient encore automatiquement de la méthode **AccorderVotreInstrument** de la classe de base **Musicien**.

Remarque Toute modification apportée à une classe de base est également apportée à toutes ses classes dérivées. Par exemple, si un champ de type **InstrumentDeMusique** était ajouté à la classe de base **Musicien**, toute classe dérivée (**Violoniste**, **Guitariste**, **Pianiste**, etc.) acquerrait alors automatiquement un champ de type **InstrumentDeMusique**. Si un bogue est introduit dans une classe de base, il devient automatiquement un bogue dans chaque classe dérivée. (Ce phénomène est connu sous l'appellation de *fragilité des classes de base*.)

Fonctionnement de l'héritage dans la programmation orientée objet

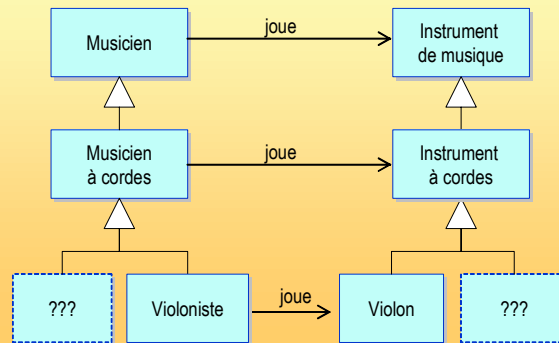
Le graphique de la diapositive représente un homme, une femme et une fillette à bicyclette. Si l'homme et la femme sont les parents biologiques de la fillette, elle héritera de la moitié de ses gènes de la part de son père et l'autre moitié de la part de sa mère.

Mais cela ne constitue pas un exemple d'héritage de classe. C'est un mécanisme d'implémentation !

Les classes sont **Homme** et **Femme**. Il existe deux instances de la classe **Femme** (une avec un attribut **age** de moins de 16) et une instance de la classe **Homme**. Il n'y a pas d'héritage de classe. Le seul moyen de trouver un héritage de classe dans cet exemple serait que la classe **Homme** et que la classe **Femme** partagent une classe de base **Personne**.

Hiérarchies des classes

■ Les classes apparentées par héritage forment des hiérarchies de classes

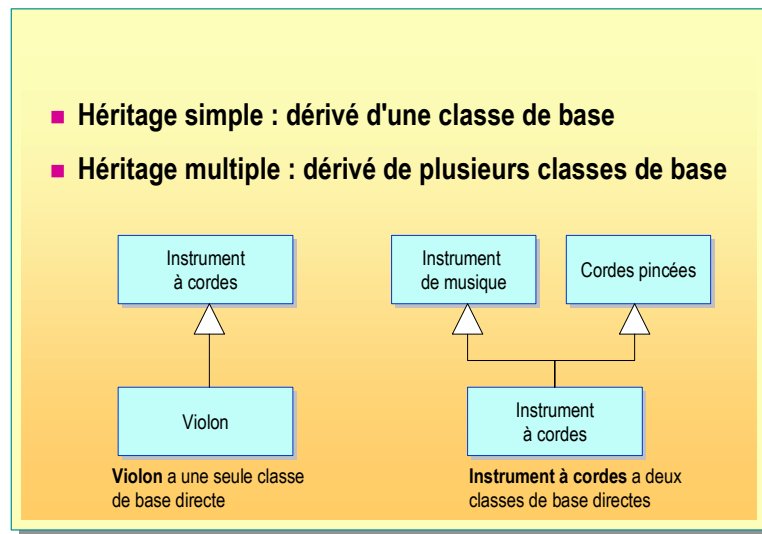


Les classes qui dérivent des classes de base peuvent elles-mêmes servir de classes de base. Par exemple, dans la diapositive, la classe **MusicienÀCordes** est dérivée de la classe **Musicien** mais sert également de classe de base à la classe dérivée **Violoniste**. Un groupe de classes apparentées par héritage forme une structure appelée *hiérarchie de classes*. Plus vous montez dans la hiérarchie, plus les classes sont des concepts généraux (généralisation). Plus vous descendez dans la hiérarchie, plus les classes sont des concepts spécialisés (spécialisation).

La profondeur d'une hiérarchie de classes se mesure au nombre de niveaux d'héritage dans la hiérarchie. Les hiérarchies de classes profondes sont plus difficiles à utiliser et à implémenter que les hiérarchies de classes superficielles. La plupart des directives de programmation recommandent de limiter la profondeur entre cinq et sept classes.

La diapositive décrit deux hiérarchies de classes parallèles : une pour les musiciens et une autre pour les instruments de musique. Il n'est pas facile de créer des hiérarchies de classes : il faut concevoir des classes en tant que classes de base dès le départ. Les hiérarchies d'héritage sont également la fonctionnalité dominante des infrastructures – modèles pouvant être employés comme bases de construction et pouvant être étendus.

Héritage simple et multiple



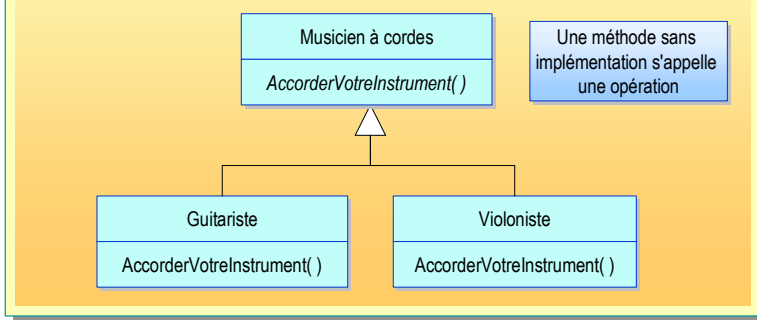
On parle d'héritage simple lorsqu'une classe possède une seule classe de base directe. Dans l'exemple de la diapositive, la classe **Violon** hérite d'une classe, **InstrumentÀCordes**, et offre un exemple d'héritage simple. **InstrumentÀCordes** dérive de deux classes, mais cela n'est pas pertinent pour la classe **Violon**. L'héritage simple peut toutefois être difficile à utiliser à bon escient. Il est bien connu que l'héritage est l'un des outils de modélisation logicielle les plus puissants qui soient, mais aussi l'un des plus mal compris et utilisés.

On parle d'héritage multiple lorsqu'une classe possède plusieurs classes de base directes. Dans l'exemple de la diapositive, la classe **InstrumentÀCordes** dérive directement de deux classes, **InstrumentDeMusique** et **CordesPincées**, et offre un exemple d'héritage multiple. L'héritage multiple offre de multiples occasions de faire un mauvais usage de l'héritage ! C#, comme la plupart des langages de programmation modernes (sauf C++), restreint l'utilisation de l'héritage multiple : Vous pouvez hériter d'autant d'interfaces que vous le souhaitez, mais vous ne pouvez hériter que d'une seule non-interface (c'est-à-dire, d'une classe abstraite ou concrète, au plus). Les termes d'interface, de classe abstraite et de classe concrète, sont traités plus loin dans ce module.

Notez que toutes les formes d'héritage, et l'héritage multiple en particulier, offrent plusieurs vues du même objet. Par exemple, un objet **Violon** peut être utilisé au niveau de la classe **Violon**, mais également au niveau de la classe **InstrumentÀCordes**.

Polymorphisme

- Le nom de la méthode réside dans la classe de base
- Les implémentations de la méthode résident dans les classes dérivées



Polymorphisme signifie littéralement *plusieurs formes*. Il s'agit du concept qu'une méthode déclarée dans une classe de base peut être implémentée de nombreuses manières différentes dans les différentes classes dérivées.

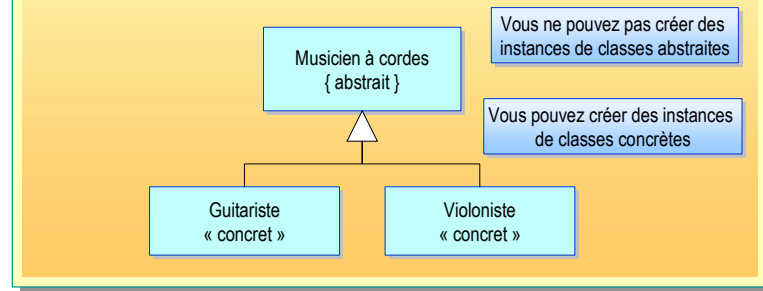
Prenons l'exemple d'un orchestre de musiciens qui accordent leur instrument avant le début du concert. Sans polymorphisme, le chef d'orchestre doit se rendre auprès de chaque musicien, identifier le type d'instrument dont il joue et lui donner des instructions précises quant à la manière d'accorder son instrument particulier. Avec polymorphisme, le chef d'orchestre peut se contenter de dire au musicien « accorde ton instrument ». Le chef d'orchestre n'a pas besoin de savoir de quel instrument joue chaque musicien. Il doit juste savoir que chaque musicien répondra à la même requête de comportement d'une manière appropriée pour son instrument. Au lieu que le chef d'orchestre détienne la connaissance de l'accordage de tous les types d'instruments différents, la connaissance est répartie entre les différents musiciens de l'orchestre : le guitariste sait accorder une guitare, le violoniste sait accorder un violon, etc. En fait, le chef d'orchestre ne sait accorder *aucun* des instruments. Cette allocation décentralisée des responsabilités signifie également que de nouvelles classes dérivées (telles que **Percussionniste**) peuvent être ajoutées à la hiérarchie sans nécessairement modifier les classes existantes (telles que le chef d'orchestre).

Il y a toutefois un problème. Quel est le corps de la méthode au niveau de la classe de base ? Sans savoir de quel type d'instrument particulier joue un musicien, il est impossible de savoir comment accorder l'instrument. Pour gérer cela, seul le nom de la méthode (et aucun corps) peut être déclaré dans la classe de base. Un nom de méthode sans corps de méthode s'appelle une *opération*. Un des moyens de dénoter une opération en UML est d'utiliser l'italique, comme dans la diapositive.

Classes de base abstraites

- Certaines classes existent dans le seul but de servir à la dérivation

- Créer des instances de ces classes n'a aucun sens
- Ces classes sont *abstraites*

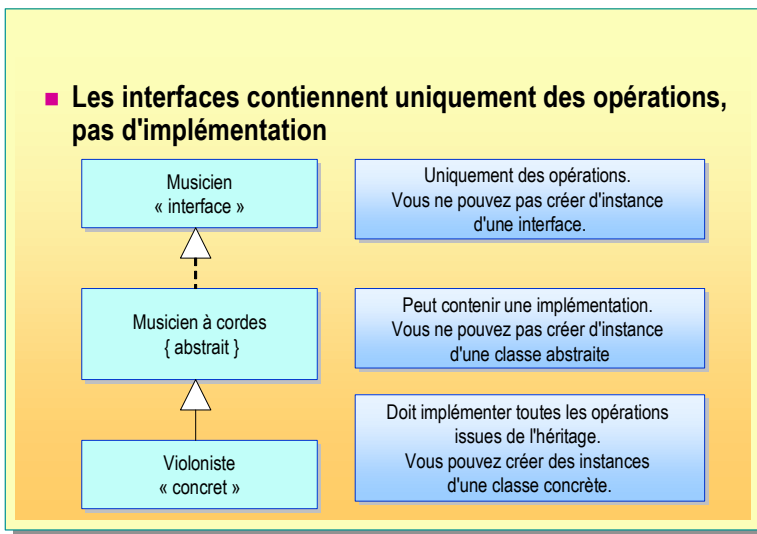


Dans une hiérarchie de classe typique, l'opération (le nom d'une méthode) est déclarée dans la classe de base, et la méthode est implémentée différemment dans chaque classe dérivée. La classe de base existe dans le seul but de présenter le nom de la méthode dans la hiérarchie. En particulier, l'opération de classe de base ne nécessite pas d'implémentation. Cela rend vital le fait que la classe de base ne soit pas utilisée comme une classe normale. Mais plus important encore, vous ne devez pas être autorisé à créer des instances de la classe de base : si vous le pouviez, que se passerait-il si vous appeliez l'opération qui n'avait pas d'implémentation ? Un mécanisme obligatoire empêche la création d'instances de ces classes de base : la classe de base doit être marquée comme abstraite.

Dans une conception UML, vous pouvez forcer une classe à être abstraite en écrivant le nom de la classe en italique ou en plaçant le mot *abstract* entre accolades ({ et }). Par contraste, vous pouvez utiliser le mot *concrete* ou *class* entre chevrons (<< et >>) comme stéréotype pour dénoter en UML une classe qui n'est pas abstraite, une classe qui peut être utilisée pour créer des instances. La diapositive illustre cela. Tous les langages de programmation orientés objet ont des constructions grammaticales qui implémentent une contrainte abstraite. (Même C++ peut utiliser des constructeurs protégés.)

La création d'une classe de base abstraite est parfois plus rétrospective : ainsi, les fonctionnalités communes en double des classes dérivées sont souvent intégrées dans une nouvelle classe de base. Toutefois, encore une fois, la classe de base doit être marquée comme abstraite car sa vocation est de servir à la création de classes dérivées, pas de créer des instances.

Interfaces



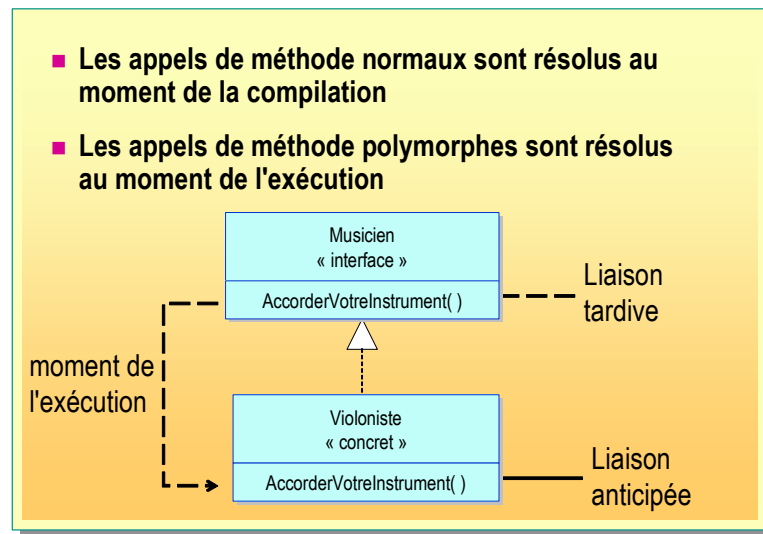
Les classes abstraites et les interfaces ont en commun de ne pas pouvoir servir à instancier des objets. Elles diffèrent toutefois dans la mesure où une classe abstraite peut contenir une implémentation tandis qu'une interface ne contient aucune implémentation de quelque sorte que ce soit ; une interface ne contient que des opérations (les noms des méthodes). On peut dire qu'une interface est encore plus abstraite qu'une classe abstraite.

En UML, vous pouvez décrire une interface en utilisant le mot interface entre chevrons (<< et >>). Tous les langages de programmation orientés objet ont des constructions grammaticales qui implémentent une interface.

Les interfaces sont des constructions importantes dans les programmes orientés objet. En UML, les interfaces ont une notation et une terminologie spécifiques. Lorsque vous dérivez d'une interface, on dit que vous *implémentez* cette interface. UML décrit ce phénomène à l'aide d'une ligne en pointillé appelée *réalisation*. Lorsque vous dérivez d'une non-interface (une classe abstraite ou une classe concrète), on dit que vous *étendez* cette classe. UML décrit ce phénomène à l'aide d'une ligne pleine appelée *généralisation/spécialisation*.

Placez vos interfaces au sommet d'une hiérarchie de classes. L'idée est simple : si vous pouvez programmer pour une interface, c'est-à-dire si vous utilisez uniquement les fonctionnalités d'un objet déclarées dans son interface, votre programme perd toute dépendance vis-à-vis de l'objet spécifique et de sa classe concrète. En d'autres termes, lorsque vous programmez pour une interface, de nombreux objets différents de nombreuses classes différentes peuvent être utilisés de manière interchangeable. C'est cette aptitude à faire des changements sans conséquences qui est à l'origine de cette maxime orientée objet : « Programme pour une interface, pas pour une implémentation ».

Liaisons anticipée et tardive



Lorsque vous initiez un appel de méthode directement sur un objet, c'est-à-dire pas via une opération de classe de base, l'appel de méthode est résolu à la compilation. On appelle également cela la *liaison anticipée* ou la *liaison statique*.

Lorsque vous initiez un appel de méthode indirectement sur un objet, c'est-à-dire, via une opération de classe de base, l'appel de méthode est résolu à l'exécution. On appelle également cela la *liaison tardive* ou la *liaison dynamique*.

Un exemple de liaison tardive se produit lorsque le chef d'orchestre indique à tous les musiciens de l'orchestre d'accorder leur instrument. En travaillant au niveau de l'interface, le chef d'orchestre n'a pas besoin de connaître (ou de dépendre de) les différents types spécifiques de musiciens concrets (comme **Violoniste**). Le chef d'orchestre est également dégagé de l'obligation de savoir qu'une nouvelle classe est ajoutée à la hiérarchie pour un nouveau type de musicien (par exemple, **Harpiste**).

La flexibilité de la liaison tardive a un coût physique et un coût logique :

- Coût physique

Les appels de liaison tardive sont légèrement plus lents que les appels de liaison anticipée. En effet, le travail supplémentaire lié à un appel de liaison tardive consiste à découvrir la classe de l'objet appelant. Ce travail est effectué de manière efficace (vous ne pourriez pas le faire plus rapidement vous-même), mais c'est un travail supplémentaire.

- Coût logique

Avec la liaison tardive, les classes dérivées peuvent être utilisées à la place de leur classe de base. Un appel d'opération peut être passé via une interface et, au moment de l'exécution, l'objet de classe dérivée verra sa méthode correctement appelée. En d'autres termes, toutes les classes dérivées qui implémentent une interface peuvent servir de substituts pour le type d'interface en question. Les nouveaux-venus à la programmation orientée objet passent souvent à côté de la possibilité de substitution qu'offre l'héritage.

Contrôle des acquis

- Classes et objets
- Utilisation de l'encapsulation
- C# et orientation objet
- Définition de systèmes orientés objet

1. Expliquez le concept d'abstraction et en quoi il est important en ingénierie logicielle.
2. Quels sont les deux principes de l'encapsulation ?

3. Décrivez l'héritage dans le cadre de la programmation orientée objet.
4. Qu'est-ce que le polymorphisme ? En quoi est-il apparenté aux liaisons anticipée et tardive ?
5. Décrivez les différences entre les interfaces, les classes abstraites et les classes concrètes.

Module 8 : Utilisation des variables de type référence

Table des matières

Vue d'ensemble	1
Utilisation des variables de type référence	2
Utilisation des types référence courants	15
Hiérarchie des objets	23
Espaces de noms du .NET Framework	29
Atelier 8.1 : Définition et utilisation des variables de type référence	35
Conversions de données	44
Présentation multimédia : Casting de type sécurisé en C#	57
Atelier 8.2 : Conversion de données	58
Contrôle des acquisitions	63



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Utilisation des variables de type référence
- Utilisation des types référence courants
- Hiérarchie des objets
- Espaces de noms du .NET Framework
- Conversions de données

Dans ce module, vous allez apprendre à utiliser les types référence en C#. Vous allez étudier plusieurs types référence, tels que le type **string**, qui sont intégrés dans le langage et l'environnement d'exécution C#. Ces éléments seront abordés en tant qu'exemples de types référence.

Vous étudierez également la hiérarchie de classes **System.Object** et le type **object** en particulier, afin de comprendre les relations existant entre les différents types référence et entre ces derniers et les types valeur. Vous apprendrez à convertir des données entre différents types référence à l'aide de conversions explicites et implicites. Vous apprendrez également à utiliser les conversions boxing et unboxing pour convertir des données entre les types référence et les types valeur.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- décrire les principales différences entre les types référence et les types valeur ;
- utiliser des types référence courants, tels que **string** ;
- expliquer comment le type **object** fonctionne et vous familiariser avec les méthodes qu'il propose ;
- décrire les espaces de noms courants du .NET Framework ;
- définir si les différents types et objets sont compatibles ;
- convertir explicitement et implicitement des types de données entre des types référence ;
- effectuer des conversions boxing/unboxing entre des références et des valeurs.

◆ Utilisation des variables de type référence

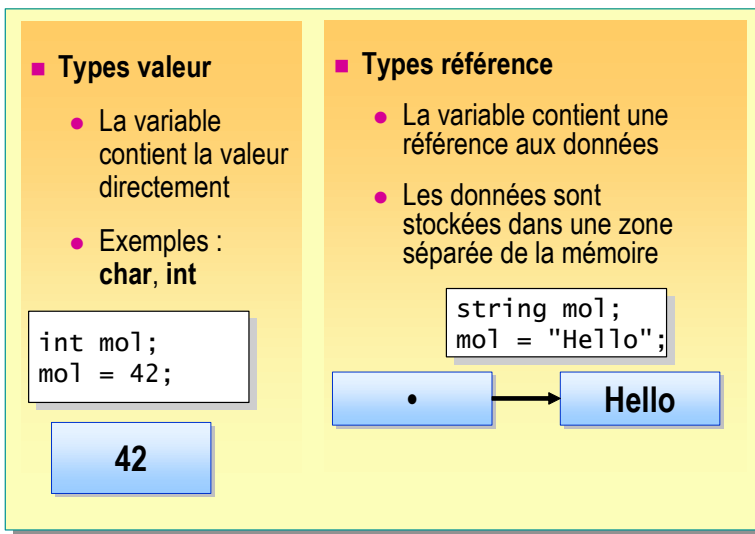
- Comparaison entre les types valeur et les types référence
- Déclaration et libération des variables référence
- Références non valides
- Comparaison de valeurs et comparaison de références
- Références multiples au même objet
- Utilisation de références comme paramètres de méthode

Les types référence sont des fonctionnalités importantes du langage C#. Ils vous permettent d'écrire des applications complexes et puissantes, et d'utiliser de manière efficace l'infrastructure d'exécution.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- décrire les principales différences entre les types référence et les types valeur ;
- utiliser et ignorer les variables référence ;
- passer des types référence en tant que paramètres de méthode.

Comparaison entre les types valeur et les types référence



Le langage C# prend en charge les types de données de base tels que **int**, **long** et **bool**. Ces types sont également appelés *types valeur*. Le langage C# prend également en charge des types de données plus complexes et plus puissants, appelés *types référence*.

Types valeur

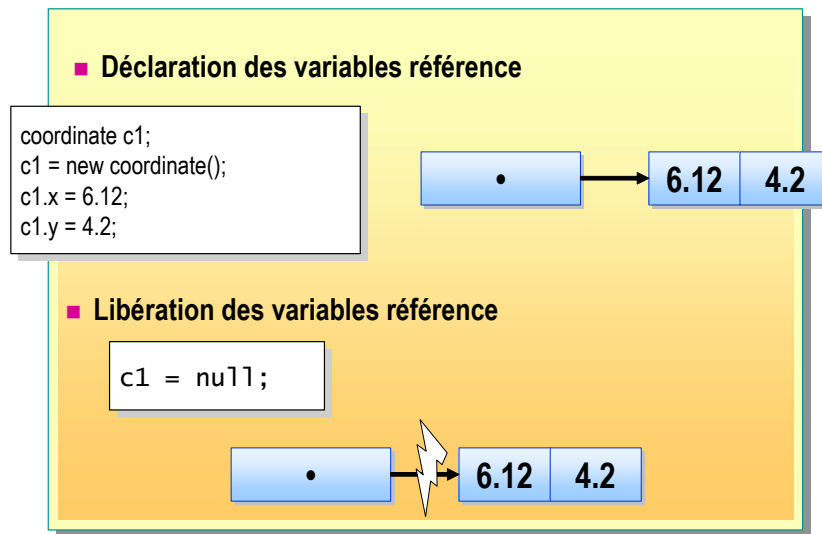
Les variables de type valeur sont les types de données intégrés de base, tels que **char** et **int**. Les types valeur sont les plus simples en C#. Les variables de type valeur contiennent leurs données directement dans la variable.

Types référence

Les variables de type référence contiennent une référence aux données, et non les données elles-mêmes. Les données sont stockées dans une zone séparée de la mémoire.

Vous avez déjà utilisé plusieurs types référence dans ce cours, peut-être sans vous en apercevoir. Les tableaux, les chaînes et les exceptions sont tous des types référence intégrés au compilateur C# et au .NET Framework. Les classes, intégrées ou définies par l'utilisateur, sont également une sorte de type référence.

Déclaration et libération des variables référence



Pour utiliser des variables de type référence, vous devez savoir comment les déclarer, les initialiser et les libérer.

Déclaration des variables référence

Pour déclarer des variables de type référence, vous devez utiliser la même syntaxe que pour déclarer des variables de type valeur :

```
coordinate c1;
```

L'exemple précédent déclare une variable *c1* pouvant contenir une référence à un objet de type **coordinate**. Toutefois, cette variable n'est pas initialisée pour référencer un objet **coordinate**.

Pour initialiser un objet **coordinate**, utilisez l'opérateur **new**. Cette opération crée un nouvel objet et retourne une référence d'objet pouvant être stockée dans la variable référence.

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

Si vous préférez, vous pouvez combiner l'opérateur **new** avec la déclaration de variable, afin que la variable soit déclarée et initialisée dans une même instruction, comme suit :

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

Après avoir créé un objet en mémoire auquel la variable *c1* fait référence, vous pouvez référencer les variables membres de cet objet à l'aide de l'opérateur **point** (**.**), comme illustré dans l'exemple suivant :

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

Exemple de déclaration de variables référence

Les classes sont des types référence. L'exemple suivant montre comment déclarer une classe définie par l'utilisateur appelée **coordinate**. Dans un souci de simplicité, cette classe comprend uniquement deux variables membres publiques : x et y .

```
class coordinate
{
    public double x = 0.0;
    public double y = 0.0;
}
```

Cette classe simple sera utilisée dans des exemples ultérieurs pour montrer comment créer, utiliser et détruire les variables référence.

Libération des variables référence

Après l'assignation d'une référence à un nouvel objet, la variable référence continue de référencer l'objet jusqu'à ce qu'elle soit assignée pour référencer un autre objet.

C# définit une valeur spéciale appelée **null**. Une variable référence contient la valeur **null** lorsqu'elle ne référence aucun objet valide. Pour libérer une référence, vous pouvez assigner explicitement la valeur **null** à une variable référence (ou tout simplement autoriser cette référence à être hors de portée).

Références non valides

- **Si vous avez des références non valides**
 - Vous ne pouvez pas accéder aux membres ou aux variables
- **Références non valides au moment de la compilation**
 - Le compilateur détecte l'utilisation de références non initialisées
- **Références non valides au moment de l'exécution**
 - Le système génère une erreur d'exception

Vous ne pouvez accéder aux membres d'un objet via une variable référence que si cette dernière a été initialisée pour désigner une référence valide. Si la référence n'est pas valide, vous ne pouvez pas accéder aux variables membres ou aux méthodes.

Le compilateur peut détecter ce problème dans certains cas. Dans d'autres cas, il doit être détecté et traité au moment de l'exécution.

Références non valides au moment de la compilation

Le compilateur peut parfois détecter qu'une variable référence n'est pas initialisée avant d'être utilisée.

Par exemple, si une variable **coordinate** est déclarée mais pas assignée, un message d'erreur semblable au suivant s'affiche : « Utilisation d'une variable locale non assignée c1. » Vous trouverez ci-dessous un exemple :

```
coordinate c1;  
c1.x = 6.12; // échouera : variable non assignée
```

Références non valides au moment de l'exécution

En général, il est impossible de déterminer au moment de la compilation si une référence variable est valide ou non. C'est pourquoi C# vérifie la valeur d'une variable référence avant son utilisation, afin de s'assurer qu'elle n'est pas égale à **null**.

Si vous tentez d'utiliser une variable référence dont la valeur est **null**, le runtime lève une exception **NullReferenceException**. Si vous le souhaitez, vous pouvez vérifier cette condition à l'aide des méthodes **try** et **catch**. En voici un exemple :

```
try {  
    c1.x = 45;  
}  
catch (NullReferenceException) {  
    Console.WriteLine("c1 a une valeur null");  
}
```

Vous pouvez également rechercher la valeur **null** explicitement, afin d'éviter les exceptions. L'exemple suivant montre comment vérifier qu'une variable référence contient une référence non null avant de tenter d'accéder à ses membres :

```
if (c1 != null)  
    c1.x = 45;  
else  
    Console.WriteLine("c1 a une valeur null");
```

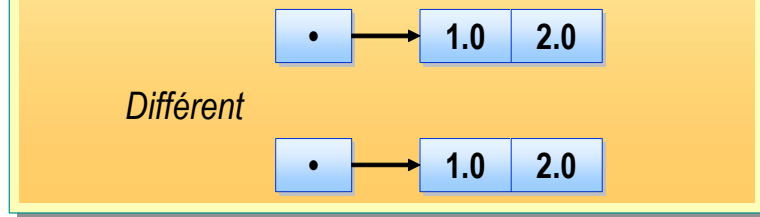
Comparaison de valeurs et comparaison de références

■ Comparaison de types valeur

- == et != comparent des valeurs

■ Comparaison de types référence

- == et != comparent les références, pas les valeurs



Les opérateurs d'égalité (==) et d'inégalité (!=) peuvent fonctionner de manière inattendue avec les variables référence.

Comparaison de types valeur

Avec les types valeur, vous pouvez utiliser les opérateurs == et != pour comparer des valeurs.

Comparaison de types référence

Avec les types référence autres que **string**, les opérateurs == et != déterminent si les deux variables référence se rapportent au même objet. Vous *ne comparez pas* le contenu des objets auxquels les variables font référence. Avec le type string, == compare la valeur des chaînes.

Examinez l'exemple suivant, dans lequel deux variables `coordinate` sont créées et initialisées sur les mêmes valeurs :

```
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("Identique");
else
    Console.WriteLine("Différent");
```

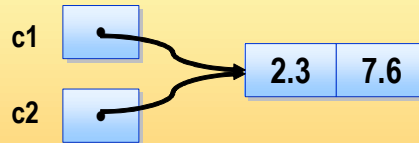
La sortie de ce code est « Différent ». Même si les objets référencés par les variables `c1` et `c2` ont la même valeur, il s'agit d'objets différents ; c'est pourquoi l'opérateur `==` retourne **false**.

Avec les types référence, vous ne pouvez pas utiliser les autres opérateurs relationnels (`<`, `>`, `<=` et `>=`) pour vérifier si deux variables font référence au même objet.

Références multiples au même objet

■ Deux références peuvent faire référence au même objet

- Deux façons d'accéder au même objet en lecture/écriture



```
coordinate c1= new coordinate( );  
coordinate c2;  
c1.x = 2.3; c1.y = 7.6;  
c2 = c1;  
Console.WriteLine(c1.x + " , " + c1.y);  
Console.WriteLine(c2.x + " , " + c2.y);
```

Comme les variables référence contiennent une référence aux données, deux variables référence peuvent se rapporter au même objet. Vous pouvez donc écrire des données via une référence et lire ces mêmes données via une autre référence.

Références multiples au même objet

Dans l'exemple ci-dessous, la variable *c1* est initialisée pour désigner une nouvelle instance de la classe, et ses variables membres *x* et *y* sont initialisées. Ensuite, la variable *c1* est copiée dans la variable *c2*. Enfin, la valeur des objets référencés par les variables *c1* et *c2* s'affiche.

```
coordinate c1 = new coordinate( );  
coordinate c2;  
c1.x = 2.3;  
c1.y = 7.6;  
c2 = c1;  
Console.WriteLine(c1.x + " , " + c1.y);  
Console.WriteLine(c2.x + " , " + c2.y);
```

La sortie de ce programme est la suivante :

```
2.3 , 7.6  
2.3 , 7.6
```

L'assignation de *c1* à *c2* copie la référence, afin que les deux variables référencent la même instance. Par conséquent, les valeurs affichées pour les variables membres de *c1* et *c2* sont identiques.

Écriture et lecture des mêmes données via des références différentes

Dans l'exemple ci-dessous, une assignation a été ajoutée immédiatement avant les appels de la méthode **Console.WriteLine**.

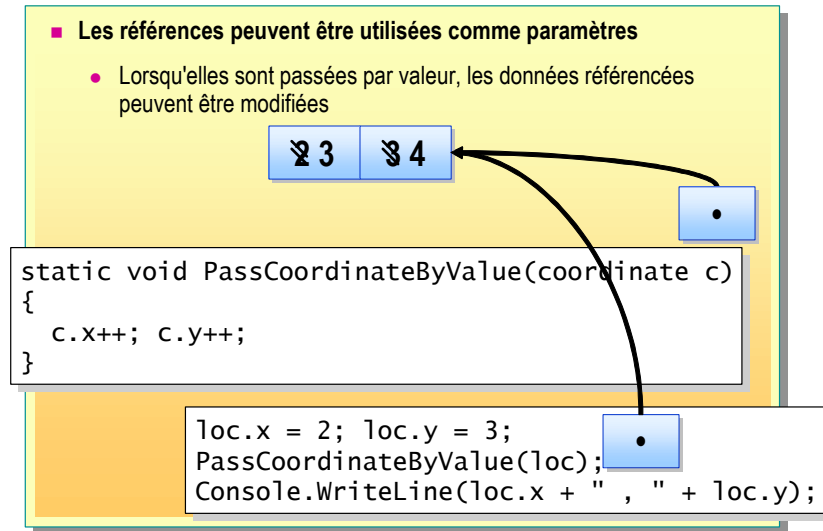
```
coordinate c1 = new coordinate( );  
coordinate c2;  
c1.x = 2.3;  
c1.y = 7.6;  
c2 = c1;  
c1.x = 99; // Il s'agit de l'instruction supplémentaire  
Console.WriteLine(c1.x + " , " + c1.y);  
Console.WriteLine(c2.x + " , " + c2.y);
```

La sortie de ce programme est la suivante :

```
99 , 7.6  
99 , 7.6
```

Cet exemple montre que l'assignation de la valeur 99 à la variable *c1.x* a également modifié la variable *c2.x*. Comme la référence de la variable *c1* a été auparavant assignée à la variable *c2*, un programme peut écrire des données via une référence et lire les mêmes données via une autre référence.

Utilisation de références comme paramètres de méthode



Vous pouvez passer des variables référence à l'intérieur et à l'extérieur d'une méthode.

Références et méthodes

Vous pouvez passer des variables référence dans des méthodes comme paramètres en utilisant l'un des trois mécanismes d'appel suivants :

- Par valeur
- Par référence
- Paramètres de sortie

L'exemple ci-dessous montre une méthode qui passe trois variables référence coordinate. La première est passée par valeur, la deuxième par référence et la troisième est un paramètre de sortie. La valeur retournée par la méthode est une variable référence coordinate.

```
static coordinate Example(
    coordinate ca,
    ref coordinate cb,
    out coordinate cc)
{
    // ...
}
```

Passage de références par valeur

Lorsque vous utilisez une variable référence en tant que paramètre de valeur, la méthode reçoit une copie de la référence. Pendant la durée de l'appel, deux références référencent donc le même objet. Par exemple, le code suivant affiche les valeurs 3 et 4 :

```
static void PassCoordinateByValue(coordinate c)
{
    c.x++; c.y++; //c référence loc
}
coordinate loc = new coordinate( );
loc.x = 2; loc.y = 3;
PassCoordinateByValue(loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

En outre, toute modification du paramètre de méthode ne peut pas affecter la référence d'appel. Par exemple, le code suivant affiche les valeurs 0 , 0 :

```
static void PassCoordinateByValue(coordinate c)
{
    c = new coordinate( ); //c ne référence plus loc
    c.x = c.y = 22.22;
}
coordinate loc = new coordinate( );
PassCoordinateByValue(loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Passage de références par référence

Lorsque vous utilisez une variable référence en tant que paramètre **ref**, la méthode reçoit la variable référence réelle. Dans ce cas, contrairement au passage par valeur, il n'y a qu'une seule référence. La méthode *ne fabrique pas* sa propre copie. Cela signifie que toute modification du paramètre de méthode affecte la référence d'appel. Par exemple, le code suivant affiche les valeurs 33,33 , 33,33 :

```
static void PassCoordinateByRef(ref coordinate c)
{
    c = new coordinate( );
    c.x = c.y = 33.33;
}
coordinate loc = new coordinate( );
PassCoordinateByRef(ref loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Passage de références par sortie

Lorsque vous utilisez une variable référence en tant que paramètre **out**, la méthode reçoit la variable référence réelle. Dans ce cas, contrairement au passage par valeur, il n'y a qu'une seule référence. La méthode *ne fabrique pas* sa propre copie. Le passage par le paramètre **out** est semblable au passage par le paramètre **ref**, sauf que la méthode *doit* être assignée au paramètre **out**. Par exemple, le code suivant affiche les valeurs 44,44 , 44,44 :

```
static void PassCoordinateByOut(out coordinate c)
{
    c = new coordinate( );
    c.x = c.y = 44.44;
}
coordinate loc = new coordinate( );
PassCoordinateByOut(out loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

Passage de références dans les méthodes

Les variables de type référence ne contiennent pas la valeur proprement dite, mais une référence à cette valeur. Cela s'applique également aux paramètres de méthode et implique que le mécanisme de passage par valeur peut produire des résultats inattendus.

En utilisant la classe **coordinate** comme exemple, examinez la méthode suivante :

```
static void PassCoordinateByValue(coordinate c)
{
    c.x++;
    c.y++;
}
```

Le paramètre *c* de la classe **coordinate** est passé par valeur. Dans cette méthode, les variables membres *x* et *y* sont incrémentées. Maintenant, étudiez le code suivant, qui appelle la méthode **PassCoordinateByValue** :

```
coordinate loc = new coordinate( );
loc.x = 2;
loc.y = 3;
PassCoordinateByValue(loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

La sortie de ce code est la suivante :

3 , 4

Cet exemple montre que les valeurs référencées par le paramètre **loc** ont été modifiées par la méthode. Cela peut sembler contradictoire avec l'explication du passage par valeur donnée précédemment dans ce cours, mais il n'en est rien. La variable référence *loc* est copiée dans le paramètre *c* et ne peut pas être modifiée par la méthode, mais la mémoire à laquelle elle se rapporte n'est pas copiée et ne subit aucune restriction de ce type. La variable *loc* fait toujours référence à la même zone de la mémoire, mais cette dernière contient maintenant des données différentes.

◆ Utilisation des types référence courants

- Classe Exception
- Classe String
- Méthodes, propriétés et opérateurs courants de la classe String
- Comparaisons de chaînes
- Opérateurs de comparaison de chaînes

Certaines classes de type référence sont intégrées au langage C#.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- décrire le fonctionnement des classes intégrées ;
- utiliser les classes intégrées en tant que modèles lors de la création de vos propres classes.

Classe Exception

- **Exception est une classe**
- **Les objets Exception sont utilisés pour lever des exceptions**
 - Créez un objet **Exception** en utilisant **new**
 - Déclenchez l'objet à l'aide de **throw**
- **Les types Exception sont des sous-classes de la classe Exception**

Vous créez et déclenchez des objets **Exception** pour lever des exceptions.

Classe Exception

Exception est le nom d'une classe fournie dans le .NET Framework.

Objets Exception

Seuls les objets de type **Exception** peuvent être déclenchés avec la méthode **throw** et interceptés avec la méthode **catch**. À cette exception près, la classe **Exception** se comporte comme les autres types référence.

Types Exception

Exception représente une erreur générique dans une application. Il existe également des types d'exceptions spécifiques (tels que **InvalidCastException**). Certaines classes qui héritent de la classe **Exception** représentent chacune de ces exceptions spécifiques.

Classe String

- Caractères multiples, données Unicode
- Nom abrégé de `System.String`
- Immuable

```
string s = "Hello";  
s[0] = 'c'; // Erreur de compilation
```

En C#, le type `string` est utilisé pour traiter des données Unicode de plusieurs caractères. (En guise de comparaison, le type `char` est un type valeur qui traite des caractères uniques.)

Le nom de type **`string`** est un nom abrégé pour la classe **`System.String`**. Le compilateur peut traiter cette forme abrégée ; **`string`** et **`System.String`** sont donc interchangeables.

La classe **`String`** représente une chaîne de caractères immuable. Une instance de **`String`** est immuable : le texte d'une chaîne ne peut pas être modifié après sa création. Les méthodes qui semblent à première vue modifier la valeur d'une chaîne retournent en fait une nouvelle instance de la chaîne qui contient la modification.

Conseil La classe **`StringBuilder`** est souvent utilisée en association avec la classe **`String`**. Une instance de **`StringBuilder`** génère une chaîne modifiable en interne qui peut être convertie en une classe **`String`** immuable une fois terminée. **`StringBuilder`** permet d'éviter la création répétée de classes **`String`** immuables temporaires et peut améliorer les performances.

La classe **`System.String`** inclut de nombreuses méthodes. Ce cours ne fournit pas de didacticiel complet pour le traitement des chaînes, mais il répertorie les méthodes les plus utiles. Pour plus d'informations, consultez les documents de l'aide du Kit de développement .NET Framework SDK.

Méthodes, propriétés et opérateurs courants de la classe String

- Crochets droits
- Méthode Insert
- Propriété Length
- Méthode Copy
- Méthode Concat
- Méthode Trim
- Méthodes ToUpper et ToLower

Crochets droits []

Vous pouvez extraire un caractère unique à un emplacement donné dans une chaîne en utilisant le nom de la chaîne suivi de l'index entre crochets droits ([et]). Cette opération est proche de l'utilisation d'un tableau. Le premier caractère de la chaîne a un index de zéro.

Vous trouverez ci-dessous un exemple :

```
string s = "Alphabet";  
char firstchar = s[2]; // 'p'
```

Les strings étant immuables, l'assignation d'un caractère à l'aide de crochets n'est pas autorisée. Toute tentative en ce sens génère une erreur de compilation, comme le montre l'exemple suivant :

```
s[2] = '*'; // Non valide
```

Méthode Insert

Si vous souhaitez insérer des caractères dans une variable string, utilisez la méthode d'instance **Insert** pour retourner une nouvelle chaîne dans laquelle une valeur particulière est insérée à un emplacement spécifié de cette chaîne. Cette méthode accepte deux paramètres : la position du début de l'insertion et la chaîne à insérer.

Vous trouverez ci-dessous un exemple :

```
string s = "C, c'est bien !";  
s = s.Insert(3, "Sharp ");  
Console.WriteLine(s); // C Sharp, c'est bien !
```


Propriété Length

La propriété **Length** retourne la longueur d'une chaîne en tant que nombre entier, comme le montre l'exemple suivant :

```
string msg = "Bonjour";  
int slen = msg.Length; // 5
```

Méthode Copy

La méthode de classe **Copy** crée une nouvelle chaîne en copiant une autre chaîne. La méthode **Copy** duplique une chaîne spécifiée.

Vous trouverez ci-dessous un exemple :

```
string s1 = "Bonjour";  
string s2 = String.Copy(s1);
```

Méthode Concat

La méthode de classe **Concat** crée une nouvelle chaîne à partir d'une ou plusieurs chaînes, ou d'un ou plusieurs objets représentés en tant que chaînes.

Vous trouverez ci-dessous un exemple :

```
string s3 = String.Concat("a", "b", "c", "d", "e", "f", "g");
```

L'opérateur + est surchargé pour les chaînes ; l'exemple ci-dessus peut donc être réécrit comme suit :

```
string s = "a" + "b" + "c" + "d" + "e" + "f" + "g";  
Console.WriteLine(s);
```

Méthode Trim

La méthode d'instance **Trim** supprime tous les caractères ou espaces spécifiés du début et de la fin d'une chaîne.

Vous trouverez ci-dessous un exemple :

```
string s = "    Bonjour    ";  
s = s.Trim( );  
Console.WriteLine(s); // "Bonjour"
```

Méthodes ToUpper et ToLower

Les méthodes d'instance **ToUpper** et **ToLower** retournent une chaîne dans laquelle tous les caractères sont convertis en majuscules et en minuscules, respectivement, comme le montre l'exemple suivant :

```
string sText = "Comment réussir ";  
Console.WriteLine(sText.ToUpper( )); // COMMENT RÉUSSIR  
Console.WriteLine(sText.ToLower( )); // comment réussir
```

Comparaisons de chaînes

- **Méthode Equals**
 - Comparaison de l'égalité de valeurs
- **Méthode Compare**
 - Autres comparaisons
 - Option permettant d'ignorer la casse
 - Classement alphabétique
- **Options de comparaison locales**

Vous pouvez utiliser les opérateurs `==` et `!=` avec les variables `string` pour comparer le contenu des chaînes.

Méthode Equals

La classe **System.String** contient une méthode d'instance appelée **Equals**, qui peut être utilisée pour comparer l'égalité de deux chaînes. Cette méthode retourne une valeur **bool** qui est **true** si les chaînes sont égales et **false** dans le cas contraire. Cette méthode est surchargée et peut être utilisée en tant que méthode d'instance ou en tant que méthode statique. L'exemple suivant montre ces deux formes.

```
string s1 = "Bienvenue";  
string s2 = "Bienvenue";  
  
if (s1.Equals(s2))  
    Console.WriteLine("Les chaînes sont identiques");  
  
if (String.Equals(s1,s2))  
    Console.WriteLine("Les chaînes sont identiques");
```

Méthode Compare

La méthode **Compare** compare deux chaînes lexicalement ; en d'autres termes, elle compare les chaînes en fonction de leur ordre de tri. La valeur retournée par la méthode **Compare** est la suivante :

- un nombre entier négatif si la première chaîne vient avant la seconde ;
- 0 si les chaînes sont identiques ;
- un nombre entier positif si la première chaîne vient après la seconde.

```
string s1 = "Tintinnabusement";  
string s2 = "Vélocipède";  
int comp = String.Compare(s1,s2); // Retour négatif
```

Par définition, toutes les chaînes, y compris les chaînes vides, renvoient une référence de comparaison supérieure à **null**, et deux références **null** renvoient une comparaison égale.

La méthode **Compare** est surchargée. Il existe une version avec trois paramètres, le troisième étant une valeur **bool** spécifiant si la casse doit être ignorée dans la comparaison. L'exemple suivant montre une comparaison qui ignore la casse :

```
s1 = "chou";  
s2 = "Chou";  
comp = String.Compare(s1, s2, true); // Ignorer la casse
```

Options de comparaison locales

La méthode **Compare** a des versions surchargées qui permettent des comparaisons de chaînes en fonction de l'ordre de tri spécifique à chaque langue. Cela peut s'avérer utile lors de l'écriture d'applications pour un marché international. La description détaillée de cette fonctionnalité dépasse la portée de ce cours. Pour plus d'informations, recherchez « System.Globalization (espace de noms) » et « CultureInfo (classe) » dans les documents de l'aide du Kit de développement .NET Framework SDK.

Opérateurs de comparaison de chaînes

- Les opérateurs `==` et `!=` sont surchargés pour les chaînes
- Ils sont équivalents à `String.Equals` et `!String.Equals`

```
string a = "Test";  
string b = "Test";  
if (a == b) ... // Retourne true
```

Les opérateurs `==` et `!=` sont surchargés pour la classe **String**. Vous pouvez les utiliser pour examiner le contenu des chaînes.

```
string a = "Test";  
string b = "Test";  
if (a == b) ... // Retourne true
```

Les opérateurs et les méthodes ci-dessous sont équivalents :

- l'opérateur `==` est équivalent à la méthode **String.Equals** ;
- l'opérateur `!=` est équivalent à la méthode **!String.Equals**.

Les autres opérateurs relationnels (`<`, `>`, `<=` et `>=`) ne sont pas surchargés pour la classe **String**.

◆ Hiérarchie des objets

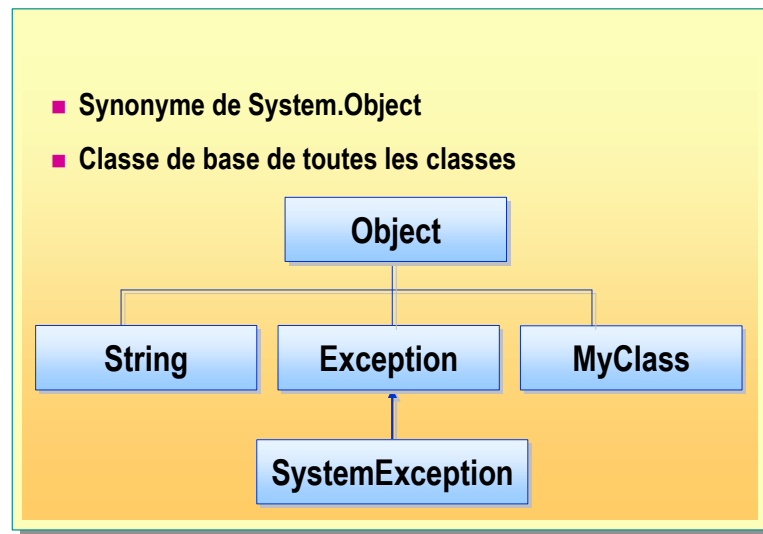
- Le type **object**
- Méthodes courantes
- Réflexion

Toutes les classes du .NET Framework dérivent de la classe **System.Object**.
Le type **object** est un alias de la classe `System.Object` en C#.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- décrire le fonctionnement de la hiérarchie d'objets ;
- utiliser le type **object**.

Le type object



Le type **object** est la classe de base de tous les types en C#.

System.Object

Le mot clé **object** est un synonyme de la classe **System.Object** dans le .NET Framework. Dès que le mot clé **object** apparaît, le nom de classe **System.Object** peut être substitué. La forme abrégée étant plus pratique, c'est la plus souvent utilisée.

Classe de base

Toutes les classes héritent d'**object**, directement ou indirectement. Cela inclut les classes que vous écrivez dans votre application et celles qui font partie de l'infrastructure système. Lorsque vous déclarez une classe sans parent explicite, vous héritez en fait d'**object**.

Méthodes courantes

■ Méthodes courantes pour tous les types référence

- Méthode **ToString**
- Méthode **Equals**
- Méthode **GetType**
- Méthode **Finalize**

Le type **object** inclut plusieurs méthodes courantes, dont tous les autres types référence héritent.

Méthodes courantes pour tous les types référence

Le type **object** fournit plusieurs méthodes courantes. Comme tous les types héritent du type **object**, les types dérivés incluent également ces méthodes. Ces méthodes courantes sont les suivantes :

- **ToString**
- **Equals**
- **GetType**
- **Finalize**

Méthode ToString

La méthode **ToString** retourne une chaîne qui représente l'objet en cours.

L'implémentation par défaut, qui se trouve dans la classe **Object**, retourne le nom du type de la classe. L'exemple suivant utilise la classe d'exemple **coordinate** définie précédemment :

```
coordinate c = new coordinate( );  
Console.WriteLine(c.ToString( ));
```

Cet exemple affiche « coordinate » sur la console.

Toutefois, vous pouvez substituer la méthode **ToString** de la classe **coordinate** pour afficher les objets de ce type sous une forme plus explicite, telle qu'une chaîne contenant les valeurs comprises dans l'objet.

Méthode Equals

La méthode **Equals** détermine si l'objet spécifié est la même instance que l'objet en cours. L'implémentation par défaut de la méthode **Equals** prend uniquement en charge l'égalité des références, comme vous l'avez vu précédemment.

Les sous-classes peuvent substituer cette méthode pour plutôt prendre en charge l'égalité des valeurs à la place.

Méthode GetType

Cette méthode permet d'extraire des informations d'exécution d'un objet. Elle est abordée plus en détail dans la section Conversions de données, plus loin dans ce module.

Méthode Finalize

Cette méthode est appelée par le runtime lorsqu'un objet devient inaccessible.

Réflexion

- Vous pouvez rechercher le type d'un objet
- Espace de noms **System.Reflection**
- L'opérateur **typeof** retourne un objet de type
 - Classes au moment de la compilation uniquement
- Méthode **GetType** de **System.Object**
 - Informations sur les classes au moment de l'exécution

Vous pouvez obtenir des informations sur le type d'un objet en utilisant le mécanisme dit de « réflexion ».

En C#, la réflexion est gérée par l'espace de noms **System.Reflection** du .NET Framework. Cet espace de noms contient des classes et des interfaces qui offrent un aperçu des types, méthodes et autres champs.

La classe **System.Type** inclut des méthodes permettant d'obtenir des informations sur la déclaration d'un type, telles que les constructeurs, les méthodes, les champs, les propriétés et les événements d'une classe. Un objet **Type** qui représente un type est unique ; en d'autres termes, deux références d'objets **Type** se rapportent au même objet uniquement si elles représentent le même type. Cela permet de comparer des objets **Type** en comparant les références (les opérateurs **==** et **!=**).

L'opérateur typeof

Au moment de la compilation, vous pouvez utiliser l'opérateur **typeof** pour retourner les informations de type à partir du nom d'un type donné.

L'exemple suivant récupère les informations relatives au type du type `byte` au moment de l'exécution et affiche le nom du type dans la console.

```
using System;
using System.Reflection;
Type t = typeof(byte);
Console.WriteLine("Type : {0}", t);
```

L'exemple suivant affiche des informations plus détaillées sur une classe. Plus précisément, il répertorie les méthodes de cette classe.

```
using System;
using System.Reflection;
Type t = typeof(string); // Lit les informations de type
MethodInfo[] mi = t.GetMethods();
foreach (MethodInfo m in mi) {
    Console.WriteLine("Méthode : {0}", m);
}
```

Méthode GetType

L'opérateur **typeof** fonctionne uniquement sur les classes qui existent au moment de la compilation. Si vous avez besoin d'informations sur le type au moment de l'exécution, vous pouvez utiliser la méthode **GetType** de la classe **Object**.

Pour plus d'informations sur la réflexion, recherchez « `system.Reflection` » dans les documents de l'aide du Kit de développement .NET Framework SDK.

◆ Espaces de noms du .NET Framework

- Espace de noms System.IO
- Espace de noms System.Xml
- Espace de noms System.Data
- Autres espaces de noms utiles

Le .NET Framework offre des services de langage courants pour différents outils de développement d'applications. Les classes du .NET Framework constituent une interface pour le Common Language Runtime, le système d'exploitation et le réseau.

Le .NET Framework est vaste et puissant, et la présentation exhaustive de l'ensemble de ses fonctionnalités dépasse la portée de ce cours. Pour plus d'informations, consultez les documents de l'aide de Microsoft Visual Studio® .NET et du Kit de développement .NET Framework SDK.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- identifier les principaux espaces de noms de l'infrastructure ;
- utiliser certains espaces de noms courants de l'infrastructure.

Espace de noms System.IO

■ Accès aux entrées/sorties du système de fichiers

- File, Directory
- StreamReader, StreamWriter
- FileStream
- BinaryReader, BinaryWriter

L'espace de noms **System.IO** est important, car il contient de nombreuses classes qui permettent à une application d'effectuer des opérations d'entrée et de sortie (E/S) de différentes manières via le système de fichiers.

Il contient également des classes qui permettent à une application d'effectuer des opérations d'entrée et de sortie sur les fichiers et les répertoires.

L'espace de noms **System.IO** est vaste et ne peut pas être expliqué en détail dans ce cours. Toutefois, la liste ci-dessous donne une idée des fonctionnalités qu'il offre :

- Les classes **File** et **Directory** permettent à une application de créer, supprimer et manipuler les répertoires et les fichiers.
- Les classes **StreamReader** et **StreamWriter** permettent à un programme d'accéder au contenu d'un fichier sous forme de flux d'octets ou de caractères.
- La classe **FileStream** peut être utilisée pour permettre l'accès aléatoire aux fichiers.
- Les classes **BinaryReader** et **BinaryWriter** permettent de lire et d'écrire des types de données primitifs en tant que valeurs binaires.

Exemple de System.IO

L'exemple suivant montre brièvement comment un fichier peut être ouvert et lu comme un flux. Il n'entend pas illustrer toutes les manières possibles d'utiliser l'espace de noms **System.IO**, mais il montre comment effectuer une simple copie de fichier.

```
using System;
using System.IO; // Utilise l'espace de noms IO
// ...
StreamReader reader = new StreamReader("infile.txt");
    // Texte à partir du fichier
StreamWriter writer = new StreamWriter("outfile.txt");
    // Texte vers le fichier
string line;
while ((line = reader.ReadLine()) != null)
{
    writer.WriteLine(line);
}

reader.Close();
writer.Close();
```

Pour ouvrir un fichier en lecture, le code de cet exemple crée un nouvel objet **StreamReader** et passe le nom du fichier à ouvrir dans le constructeur. De même, pour ouvrir un fichier en écriture, cet exemple crée un nouvel objet **StreamWriter** et passe le nom du fichier de sortie dans le constructeur. Dans cet exemple, les noms des fichiers sont codés de manière irréversible, mais il peut également s'agir de variables string.

Le programme exemple copie un fichier en lisant une ligne à la fois à partir du flux d'entrée et écrit cette ligne dans le flux de sortie.

ReadLine et **WriteLine** peuvent vous être familières. La classe **Console** comprend deux méthodes statiques de ce nom. Dans cet exemple, les méthodes sont des méthodes d'instance des classes **StreamReader** et **StreamWriter**, respectivement.

Pour plus d'informations sur l'espace de noms **System.IO**, recherchez « System.IO (espace de noms) » dans les documents de l'aide du Kit de développement .NET Framework SDK.

Espace de noms System.Xml

- Prise en charge de XML
- Plusieurs normes XML

Les applications qui doivent interagir avec le langage XML (Extensible Markup Language) peuvent utiliser l'espace de noms **System.Xml**, qui offre une prise en charge basée sur les normes du traitement XML.

L'espace de noms **System.Xml** prend en charge plusieurs normes XML, notamment les suivantes :

- XML 1.0 avec prise en charge de la définition de type de document (DTD) ;
- Espaces de noms XML
- Schémas XSD
- Expressions Xpath
- Transformations XSL/T
- Modèle DOM (Noyau) Niveau 1
- Modèle DOM (Noyau) Niveau 2

La classe **XmlDocument** est utilisée pour représenter un document XML entier. Les éléments et les attributs d'un document XML sont représentés dans les classes **XmlElement** et **XmlAttribute**.

La description détaillée des espaces de noms XML dépasse la portée de ce cours. Pour plus d'informations, recherchez « System.Xml (espace de noms) » dans les documents de l'aide du Kit de développement .NET Framework SDK.

Espace de noms System.Data

- **System.Data.SqlClient**
 - Fournisseur de données SQL Server .NET
- **System.Data**
 - Principalement constitué de classes qui forment l'architecture ADO.NET

L'espace de noms **System.Data** est constitué de classes qui forment l'architecture ADO.NET. Celle-ci vous permet de générer des composants qui gèrent efficacement les données à partir de plusieurs sources de données. ADO.NET contient des outils permettant d'interroger, mettre à jour et réconcilier les données de systèmes *n*-couches.

Dans l'architecture ADO.NET, vous pouvez utiliser la classe **DataSet**. Chaque classe **DataSet** contient des objets **DataTable**, et chaque objet **DataTable** contient des données provenant d'une source de données unique, telle que Microsoft SQL Server™.

L'espace de noms **System.Data.SqlClient** permet d'accéder directement à SQL Server. Notez que cet espace de noms est propre à SQL Server.

Pour accéder aux autres bases de données relationnelles et sources de données structurées, vous pouvez utiliser l'espace de noms **System.Data.OleDb**, qui offre un accès optimal aux pilotes de base de données OLEDB.

La description détaillée des espaces de noms **System** dépasse la portée de ce cours. Pour plus d'informations, recherchez « System.Data (espace de noms) » dans les documents de l'aide du Kit de développement .NET Framework SDK.

Autres espaces de noms utiles

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

Le .NET Framework comporte de nombreux autres espaces de noms et classes. Ce cours ne les aborde pas en détail, mais les informations ci-dessous vous aideront lorsque vous rechercherez des informations dans les fichiers de référence et dans la documentation :

- L'espace de noms **System** contient des classes qui définissent les types de données valeur et référence, les événements et gestionnaires d'événements, les interfaces, les attributs et les exceptions de traitement couramment utilisés. D'autres classes fournissent des services qui prennent en charge les conversions des types de données, la manipulation des paramètres de méthode, les opérations mathématiques, l'appel de programmes à distance et en local et la gestion d'applications.
- L'espace de noms **System.Net** inclut une interface de programmation simple pour la plupart des protocoles réseau actuels. L'espace de noms **System.Net.Sockets** offre une implémentation de l'interface Microsoft Windows® Sockets aux développeurs ayant besoin d'un accès de bas niveau aux services réseau TCP/IP (Transmission Control Protocol/Internet Protocol).
- **System.Windows.Forms** est l'infrastructure de l'interface utilisateur graphique (GUI) des applications Windows ; il prend en charge les formulaires, les contrôles et les gestionnaires d'événements.

Pour plus d'informations sur les espaces de noms **System**, recherchez « System (espace de noms) » dans les documents de l'aide du Kit de développement Microsoft .NET Framework SDK.

Atelier 8.1 : Définition et utilisation des variables de type référence



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer des variables référence et les passer en tant que paramètres de méthode ;
- utiliser les infrastructures système.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les éléments suivants :

- création et utilisation de classes ;
- appel de méthodes et passage de paramètres ;
- utilisation de tableaux.

Durée approximative de cet atelier : 45 minutes

Exercice 1

Ajout d'une méthode d'instance avec deux paramètres

Dans l'atelier 7, vous avez développé une classe **BankAccount**. Dans cet exercice, vous allez réutiliser cette classe et ajouter une nouvelle méthode d'instance, appelée **TransferFrom**, qui permet de transférer de l'argent d'un compte donné à celui-ci. Si vous n'avez pas terminé l'atelier 7, vous pouvez obtenir une copie de la classe **BankAccount** dans le dossier *dossier d'installation*\Labs\Lab08\Starter\Bank.

► Pour créer la méthode **TransferFrom**

1. Ouvrez le projet Bank.sln situé dans le dossier *dossier d'installation*\Labs\Lab08\Starter\Bank.
2. Modifiez la classe **BankAccount** comme suit :
 - a. Créez une méthode d'instance publique appelée **TransferFrom** dans la classe **BankAccount**.
 - b. Le premier paramètre est une référence à un autre objet **BankAccount**, appelé **accFrom**, à partir duquel l'argent doit être transféré.
 - c. Le second paramètre est une valeur de type **decimal**, appelée *amount*, passée par valeur et indiquant la somme à transférer.
 - d. La méthode n'a pas de valeur de retour.
3. Dans le corps de la méthode **TransferFrom**, ajoutez deux instructions qui exécutent les tâches suivantes :
 - a. Débiter *amount* du solde de **accFrom** (à l'aide de **Withdraw**).
 - b. Effectuer un test pour vérifier que le retrait a réussi. Si tel est le cas, créditer *amount* sur le solde du compte courant (à l'aide de **Deposit**).

La classe **BankAccount** doit ressembler à ce qui suit :

```
class BankAccount
{
    ... code supplémentaire omis pour des raisons de clarté
    ...

    public void TransferFrom(BankAccount accFrom, decimal
↪amount)
    {
        if (accFrom.Withdraw(amount))
            this.Deposit(amount);
    }
}
```

► Pour tester la méthode **TransferFrom**

1. Ajoutez le fichier **Test.cs** au projet en cours.
2. Ajoutez la classe suivante à ce fichier. Il s'agit du test de validation.

```
using System;

public class Test
{
    public static void Main()
    {

    }
}
```

3. Dans la méthode **Main**, ajoutez du code pour créer deux objets **BankAccount**, chacun ayant un solde initial de 100 (utilisez la méthode **Populate**).
4. Ajoutez du code pour afficher le type, le numéro et le solde actuel de chaque compte.
5. Ajoutez du code pour appeler la méthode **TransferFrom** et transférer 10 d'un compte à l'autre.
6. Ajoutez du code pour afficher les soldes actuels une fois le transfert effectué.

La classe **Test** peut ressembler à ce qui suit :

```
public static void Main()
{
    BankAccount b1 = new BankAccount( );
    b1.Populate(100);

    BankAccount b2 = new BankAccount( );
    b2.Populate(100);

    Console.WriteLine("Avant le transfert");
    Console.WriteLine("{0} {1} {2}",
        ↪b1.Type( ), b1.Number( ), b1.Balance( ));
    Console.WriteLine("{0} {1} {2}",
        ↪b2.Type( ), b2.Number( ), b2.Balance( ));

    b1.TransferFrom(b2, 10);

    Console.WriteLine("Après le transfert");
    Console.WriteLine("{0} {1} {2}",
        ↪b1.Type( ), b1.Number( ), b1.Balance( ));
    Console.WriteLine("{0} {1} {2}",
        ↪b2.Type( ), b2.Number( ), b2.Balance( ));
}
```

7. Enregistrez votre travail.
8. Compilez le projet et corrigez les erreurs. Exécutez et testez le programme.

Exercice 2

Inversion d'une chaîne

Dans le module 5, vous avez développé une classe **Utils** qui contenait différentes méthodes utilitaires.

Dans cet exercice, vous allez ajouter une nouvelle méthode statique appelée **Reverse** à la classe **Utils**. Cette méthode prend une chaîne et retourne une nouvelle chaîne dans laquelle les caractères figurent dans l'ordre inverse.

► Pour créer la méthode **Reverse**

1. Ouvrez le projet `Utils.sln` situé dans le dossier *dossier d'installation*\Labs\Lab08\Starter\Utils.
2. Ajoutez une méthode statique publique appelée **Reverse** à la classe **Utils**, comme suit :
 - a. La méthode contient un paramètre unique appelé *s* qui référence une valeur de type **string**.
 - b. Elle a une valeur de retour **void**.
3. Dans la méthode **Reverse**, créez une variable **string** appelée *sRev* pour contenir la chaîne retournée. Initialisez cette chaîne à "".
4. Pour créer une chaîne inversée :
 - a. Écrivez une boucle pour extraire un caractère à la fois du paramètre *s*. Commencez à la fin (utilisez la propriété **Length**), puis remontez jusqu'au début de la chaîne. Vous pouvez utiliser une notation de tableau (`[]`) pour examiner un caractère individuel au sein d'une chaîne.

Conseil Le dernier caractère d'une chaîne se trouve à l'emplacement **Length – 1**. Le premier caractère se trouve à l'emplacement 0.

- b. Ajoutez ce caractère à la fin de *sRev*.

La classe **Utils** peut contenir les éléments suivants :

```
class Utils
{
    ... autres méthodes omises pour plus de clarté ...

    //
    // Inverse une chaîne
    //

    public static void Reverse(ref string s)
    {
        string sRev = "";

        for (int k = s.Length - 1; k >= 0 ; k--)
            sRev = sRev + s[k];

        // Retourne le résultat à l'appelant
        s = sRev;
    }
}
```

► **Pour tester la méthode Reverse**

1. Ajoutez le fichier Test.cs au projet en cours.
2. Ajoutez la classe suivante à ce fichier. Il s'agit du test de validation.

```
namespace Utils
{
    using System;

    public class Test
    {
        public static void Main()
        {
        }
    }
}
```

3. Dans la méthode **Main**, créez une variable **string**.
4. Lisez une valeur dans la variable **string** à l'aide de la méthode **Console.ReadLine**.
5. Passez la chaîne dans la méthode **Reverse**. N'oubliez pas le mot clé **ref**.
6. Affichez la valeur retournée par la méthode **Reverse**.

La classe **Test** peut contenir les éléments suivants :

```
static void Main( )
{
    string message;

    // Lit une chaîne d'entrée
    Console.WriteLine("Entrez la chaîne à inverser : ");
    message = Console.ReadLine( );

    // Inverse la chaîne
    Utils.Reverse(ref message);

    // Affiche le résultat
    Console.WriteLine(message);
}
```

7. Enregistrez votre travail.
8. Compilez le projet et corrigez les erreurs. Exécutez et testez le programme.

Exercice 3

Création d'une copie d'un fichier en majuscules

Dans cet exercice, vous allez écrire un programme invitant l'utilisateur à entrer le nom d'un fichier texte. Le programme vérifiera que ce fichier existe. Si tel n'est pas le cas, il affichera un message et se fermera. Le fichier sera ouvert et copié dans un autre fichier (le programme invitera l'utilisateur à donner le nom de ce fichier), mais tous les caractères seront convertis en majuscules.

Avant de commencer, il est recommandé de consulter brièvement la documentation relative à l'espace de noms **System.IO** dans les documents de l'aide du Kit de développement .NET Framework SDK. Consultez plus particulièrement la documentation relative aux classes **StreamReader** et **StreamWriter**.

► Pour créer l'application de copie du fichier

1. Ouvrez le projet CopyFileUpper.sln situé dans le dossier *dossier d'installation\Labs\Lab08\Starter\CopyFileUpper*.
2. Modifiez la classe **CopyFileUpper** et ajoutez une instruction **using** pour l'espace de noms **System.IO**.
3. Dans la méthode **Main**, déclarez deux variables **string** appelées *sFrom* et *sTo* pour contenir les noms des fichiers d'entrée et de sortie.
4. Déclarez une variable de type **StreamReader** appelée *srFrom*. Cette variable doit contenir la référence au fichier d'entrée.
5. Déclarez une variable de type **StreamWriter** appelée *swTo*. Cette variable doit contenir la référence au flux de sortie.
6. Demandez le nom du fichier d'entrée, lisez ce nom et stockez-le dans la variable de type **string** appelée *sFrom*.
7. Demandez le nom du fichier de sortie, lisez ce nom et stockez-le dans la variable de type **string** appelée *sTo*.
8. Les opérations d'entrée/sortie que vous allez utiliser peuvent lever des exceptions ; vous devez donc commencer un bloc **try-catch** pouvant intercepter une exception **FileNotFoundException** (pour les fichiers inexistants) et **Exception** (pour toutes les autres exceptions). Affichez un message explicite pour chaque exception.
9. Dans le bloc **try**, créez un nouvel objet **StreamReader** à l'aide du nom du fichier d'entrée défini dans *sFrom*, et stockez-le dans la variable référence **StreamReader** appelée *srFrom*.
10. De même, créez un nouvel objet **StreamWriter** à l'aide du nom du fichier d'entrée défini dans *sTo*, et stockez-le dans la variable référence **StreamWriter** appelée *swTo*.
11. Ajoutez une boucle **while** pour vérifier que la méthode **Peek** du flux d'entrée ne retourne pas la valeur -1. Dans cette boucle :
 - a. Utilisez la méthode **ReadLine** sur le flux d'entrée pour lire la ligne d'entrée suivante dans une variable de type **string** appelée *sBuffer*.
 - b. Exécutez la méthode **ToUpper** sur la variable *sBuffer*.
 - c. Utilisez la méthode **WriteLine** pour envoyer la variable *sBuffer* au flux de sortie.

12. Une fois la boucle terminée, fermez les flux d'entrée et de sortie.

13. Le fichier CopyFileUpper.cs doit ressembler à ce qui suit :

```
using System;
using System.IO;

class CopyFileUpper
{
    static void Main( )
    {
        string      sFrom, sTo;
        StreamReader srFrom;
        StreamWriter swTo;

        // Demande le nom du fichier d'entrée
        Console.Write("Copier de :");
        sFrom = Console.ReadLine( );

        // Demande le nom du fichier de sortie
        Console.Write("Copier vers :");
        sTo = Console.ReadLine( );

        Console.WriteLine("Copier de {0} vers {1}", sFrom,
            ↪sTo);

        try
        {
            srFrom = new StreamReader(sFrom);
            swTo    = new StreamWriter(sTo);

            while (srFrom.Peek( )!=-1)
            {
                string sBuffer = srFrom.ReadLine( );
                sBuffer = sBuffer.ToUpper( );
                swTo.WriteLine(sBuffer);
            }
            swTo.Close( );
            srFrom.Close( );

        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("Fichier d'entrée
                ↪introuvable");
        }
        catch(Exception e)
        {
            Console.WriteLine("Exception inattendue");
            Console.WriteLine(e.ToString( ));
        }
    }
}
```

14. Enregistrez votre travail. Compilez le projet et corrigez les erreurs.

► Pour tester le programme

1. Ouvrez une fenêtre Commande et accédez au dossier *dossier d'installation\Labs\Lab08\Starter\CopyFileUpper\bin\debug*.
2. Exécutez CopyFileUpper.
3. À l'invite, spécifiez le nom du fichier source
lecteur:\chemin\CopyFileUpper.cs
(Il s'agit du fichier source que vous venez de créer.)
4. Spécifiez un fichier de destination **Test.cs**.
5. Une fois le programme terminé, utilisez un éditeur de texte pour examiner le fichier Test.cs. Ce fichier doit contenir une copie de votre code source entièrement en majuscules.

◆ Conversions de données

- Conversion des types valeur
- Conversions parents/enfants
- L'opérateur **is**
- L'opérateur **as**
- Conversions et type **object**
- Conversions et interfaces
- Boxing et unboxing

Cette section explique comment effectuer des conversions de données entre des types référence en C#. Vous pouvez convertir des références d'un type à un autre, mais les types référence doivent être apparentés.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- identifier les conversions autorisées et interdites entre les types référence ;
- utiliser les mécanismes de conversion (casts, **is** et **as**) ;
- identifier les aspects particuliers de la conversion vers et depuis le type **object** ;
- utiliser le mécanisme de réflexion, qui permet d'examiner les informations d'exécution ;
- effectuer des conversions automatiques (boxing et unboxing) entre les types valeur et les types référence.

Conversion des types valeur

- Conversions implicites
- Conversions explicites
 - Opérateur de cast
- Exceptions
- Classe `System.Convert`
 - Gère les conversions en interne

C# prend en charge les conversions de données implicites et explicites.

Conversions implicites

Pour les types valeur, vous avez étudié deux modes de conversion des données : la conversion implicite et la conversion explicite à l'aide de l'opérateur de cast.

La conversion implicite a lieu lorsqu'une valeur d'un type donné est assignée à un autre type. C# autorise uniquement la conversion implicite pour certaines combinaisons de types, habituellement lorsque la première valeur peut être convertie dans la seconde sans aucune perte de données. L'exemple suivant montre comment les données sont converties de manière implicite de **int** en **long** :

```
int a = 4;
long b;
b = a; // Conversion implicite de int en long
```

Conversions explicites

Vous pouvez convertir des types valeur de manière explicite à l'aide de l'opérateur de cast, comme le montre l'exemple suivant :

```
int a;
long b = 7;
a = (int) b;
```

Exceptions

Lorsque vous utilisez l'opérateur de cast, notez que vous pouvez rencontrer certains problèmes si la valeur ne peut pas être maintenue dans la variable cible. Si un problème est détecté au cours d'une conversion explicite (par exemple, si vous tentez de placer la valeur 9 999 999 999 dans une variable **int**), C# peut lever une exception (dans ce cas, **OverflowException**). Si vous le souhaitez, vous pouvez intercepter cette exception à l'aide de **try** et **catch**, comme suit :

```
try {  
    a = checked((int) b);  
}  
catch (Exception) {  
    Console.WriteLine("Problème dans le cast");  
}
```

Pour les opérations impliquant des nombres entiers, utilisez le mot clé **checked** ou utilisez les paramètres de compilation appropriés ; sinon, la vérification ne sera pas effectuée.

Classe System.Convert

Au sein du .NET Framework, les conversions entre les différents types de base (tels que **int**, **long** et **bool**) sont gérées par la classe **System.Convert**.

Normalement, vous n'avez pas à appeler les méthodes de la classe **System.Convert**. Le compilateur gère ces appels automatiquement.

Conversions parents/enfants

- **Conversion en référence de classe parente**
 - Implicite ou explicite
 - Réussit toujours
 - Peut toujours assigner à un objet
- **Conversion en référence de classe enfant**
 - Conversion explicite à l'aide d'un cast requise
 - Vérifie que le type de la référence est correct
 - Lève une exception **InvalidCastException** si ce n'est pas le cas

Vous pouvez convertir une référence à un objet d'une classe enfant en un objet de sa classe parente, et vice-versa, sous certaines conditions.

Conversion en référence de classe parente

Les références aux objets d'un type de classe peuvent être converties en références d'un autre type si une classe hérite de l'autre, directement ou indirectement.

Une référence à un objet peut toujours être convertie en une référence à un objet de classe parente. Cette conversion peut être effectuée de manière implicite (par assignation ou au sein d'une expression) ou de manière explicite (à l'aide de l'opérateur de cast).

Les exemples suivants utilisent deux classes : **Animal** et **Bird**. **Animal** est la classe parente de **Bird** ; en d'autres termes, **Bird** hérite de **Animal**.

L'exemple suivant déclare une variable de type **Animal** et une variable de type **Bird** :

```
Animal a;  
Bird b = new Bird(...);
```

Maintenant, étudiez l'assignation suivante, dans laquelle la référence de *b* est copiée dans *a* :

```
a = b;
```

La classe **Bird** hérite de la classe **Animal**. Par conséquent, une méthode qui se trouve dans **Animal** figure également dans **Bird**. (Il se peut que la classe **Bird** ait substitué certaines méthodes de la classe **Animal** pour créer sa propre version de ces méthodes, mais une implémentation de ces méthodes existe néanmoins.) Par conséquent, les références aux objets **Bird** peuvent être assignées aux variables contenant des références aux objets de type **Animal**.

Dans ce cas, C# effectue une conversion de type **Bird** en **Animal**. Vous pouvez convertir **Bird** en **Animal** de manière explicite à l'aide de l'opérateur de cast, comme le montre l'exemple suivant :

```
a = (Animal) b;
```

Le code précédent produit exactement le même résultat.

Conversion en référence de classe enfant

Vous pouvez convertir une référence en un type enfant, mais vous devez spécifier explicitement la conversion à l'aide d'un cast. La compatibilité des types est vérifiée au moment de l'exécution pour les conversions explicites, comme le montre l'exemple suivant :

```
Bird b = (Bird) a; // Ok
```

Ce code sera correctement compilé. Au moment de l'exécution, l'opérateur de cast effectue une vérification afin de déterminer si l'objet référencé est réellement de type **Bird**. Si ce n'est pas le cas, l'exception d'exécution **InvalidCastException** est levée.

Si vous tentez d'assigner une référence à un type enfant sans opérateur de conversion, comme dans le code suivant, le compilateur affiche un message d'erreur indiquant « Impossible de convertir implicitement le type 'Animal' en 'Bird'. »

```
b = a; // ne sera pas compilé
```

Vous pouvez intercepter une erreur de conversion de types à l'aide de **try** et **catch**, comme pour toute autre exception, comme le montre le code suivant :

```
try {  
    b = (Bird) a;  
}  
catch (InvalidCastException) {  
    Console.WriteLine("N'est pas de type bird");  
}
```

L'opérateur is

■ Retourne true si une conversion est possible

```
Bird b;  
if (a is Bird)  
    b = (Bird) a; // Sûr  
else  
    Console.WriteLine("N'est pas de type Bird");
```

Vous pouvez gérer des types incompatibles en interceptant la méthode **InvalidCastException**. Il existe cependant d'autres manières de procéder, telles que l'utilisation de l'opérateur **is**.

Vous pouvez utiliser l'opérateur **is** pour tester le type de l'objet sans effectuer de conversion. L'opérateur **is** retourne **true** si la valeur de gauche n'est pas **null**, et un cast de la classe de droite ne lève pas d'exception. Dans le cas contraire, **is** retourne **false**.

```
if (a is Bird)  
    b = (Bird) a; // Sûr, car "a is Bird" retourne true  
else  
    Console.WriteLine("N'est pas de type Bird");
```

Vous pouvez considérer la relation entre les classes héritées comme une relation « est une espèce de », comme dans « Un oiseau est une espèce d'animal. » Les références de la variable *a* doivent se rapporter à des objets **Animal**, et *b* est une espèce d'animal. Bien entendu, *b* est également un oiseau, mais un oiseau n'est qu'une espèce d'animal particulier. L'inverse n'est pas vrai. Un animal n'est pas un type d'oiseau. Certains animaux sont des oiseaux, mais tous les animaux ne sont pas des oiseaux.

L'expression suivante peut donc être lue comme « Si *a* est une espèce d'oiseau » ou « Si *a* est un oiseau ou une espèce dérivée de bird ».

```
if (a is Bird)
```

L'opérateur as

- Permet de convertir des types référence, comme l'opérateur de cast
- En cas d'erreur
 - Retourne null
 - Ne lève pas d'exception

```
Bird b = a as Bird; // Conversion  
  
if (b == null)  
    Console.WriteLine("N'est pas de type bird");
```

Vous pouvez utiliser l'opérateur **as** pour effectuer des conversions entre types.

Exemple

L'instruction suivante convertit la référence dans *a* en une valeur qui référence une classe de type **Bird**, et le runtime vérifie automatiquement que cette conversion est acceptable.

```
b = a as Bird;
```

Gestion des erreurs

Contrairement à l'opérateur de cast, l'opérateur **is** gère les erreurs. Si, dans l'exemple précédent, la référence dans la variable *a* ne peut pas être convertie en une référence à un objet de la classe **Bird**, la valeur **null** est stockée dans la variable *b*, et le programme continue. L'opérateur **as** ne lève jamais d'exception.

Vous pouvez réécrire le code précédent de la manière suivante pour afficher un message d'erreur si la conversion est impossible :

```
Bird b = a as Bird;  
if (b == null)  
    Console.WriteLine("N'est pas de type bird");
```

Même si **as** ne lève jamais d'exception, toute tentative d'accès via la valeur convertie lève une exception **NullReferenceException** si cette valeur est **null**. En conséquence, il est recommandé de toujours vérifier la valeur de retour de l'opérateur **as**.

Conversions et type object

- Le type **object** est la base de toutes les classes
- Toute référence peut être assignée au type **object**
- Toute variable **object** peut être assignée à une référence
 - Conversion de types et vérifications appropriées
- Le type **object** et l'opérateur **is**

```
object ox;  
ox = a;  
ox = (object) a;  
ox = a as object;
```

```
b = (Bird) ox;  
b = ox as Bird;
```

Tous les types référence sont fondés sur le type **object**. Par conséquent, toute référence peut être stockée dans une variable de type **object**.

Le type **object** est la base de toutes les classes

Le type **object** est la base de tous les types référence.

Toute référence peut être assignée au type **object**

Comme toutes les classes sont directement ou indirectement fondées sur le type **object**, vous pouvez assigner n'importe quelle référence à une variable de type **object**, soit au moyen d'une conversion implicite, soit à l'aide d'un cast. Vous trouverez ci-dessous un exemple :

```
object ox;  
ox = a;  
ox = (object) a;  
ox = a as object;
```

Toute variable **object** peut être assignée à une référence

Vous pouvez assigner une valeur de type **object** à n'importe quelle référence d'objet, si vous effectuez correctement le cast. Notez que le runtime effectue un test afin de vérifier que la valeur assignée est du type approprié. Vous trouverez ci-dessous un exemple :

```
b = (Bird) ox;  
b = ox as Bird;
```

Les exemples précédents peuvent être écrits avec une vérification complète des erreurs, comme suit :

```
try {  
    b = (Bird) ox;  
}  
catch (InvalidCastException) {  
    Console.WriteLine("Conversion en Bird impossible");  
}  
b = ox as Bird;  
if (b == null)  
    Console.WriteLine("Conversion en Bird impossible");
```

Le type object et l'opérateur is

Comme toute valeur est en définitive dérivée du type **object**, le test d'une valeur avec l'opérateur **is** pour vérifier qu'il s'agit d'un type **object** retourne toujours **true**.

```
if (a is object) // Retourne toujours true
```

Conversion et interfaces

- Une interface peut uniquement être utilisée pour accéder à ses propres membres
- Les autres méthodes et variables de la classe ne sont pas accessibles via l'interface

Vous pouvez effectuer des conversions à l'aide des opérateurs de cast, **as** et **is**, lorsque vous travaillez avec des interfaces.

Par exemple, vous pouvez déclarer une variable de type interface, comme suit :

```
IHashCodeProvider hcp;
```

Conversion d'une référence en une interface

Vous pouvez utiliser l'opérateur de cast pour convertir la référence d'objet en une référence à une interface donnée, comme suit :

```
IHashCodeProvider hcp;  
hcp = (IHashCodeProvider) x;
```

Comme pour la conversion entre les références de classes, l'opérateur de cast lève une exception **InvalidCastException** si l'objet spécifié n'implémente pas l'interface. Vous devez déterminer si un objet prend en charge une interface avant d'effectuer un cast de cet objet, ou bien utiliser les opérateurs **try** et **catch** pour intercepter l'exception.

Détermination de l'implémentation d'une interface

Vous pouvez utiliser l'opérateur **is** pour déterminer si un objet prend en charge une interface. La syntaxe est identique à celle utilisée pour les classes :

```
if (x is IHashCodeProvider) ...
```

Utilisation de l'opérateur **as**

Vous pouvez également utiliser l'opérateur **as** à la place du casting, comme suit :

```
HashCodeProvider hcp;  
hcp = x as IHashCodeProvider;
```

Comme pour la conversion entre les classes, si la référence convertie ne prend pas en charge l'interface, l'opérateur **as** retourne **null**.

Après avoir converti la référence d'une classe en référence à une interface, la nouvelle référence peut uniquement accéder aux membres de cette interface, et non aux autres membres publics de la classe.

Exemple

Examinez l'exemple suivant pour apprendre comment fonctionne la conversion des références en interfaces. Supposons que vous avez créé une interface appelée **IVisual** qui spécifie une méthode appelée **Paint**, comme suit :

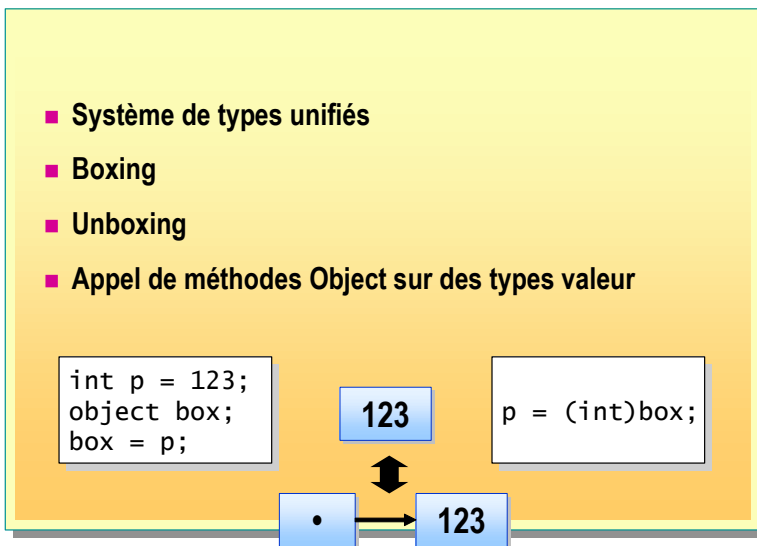
```
interface IVisual  
{  
    void Paint( );  
}
```

Supposons que vous avez également une classe **Rectangle** qui implémente l'interface **IVisual**. Elle implémente la méthode **Paint**, mais elle peut également définir ses propres méthodes. Dans cet exemple, **Rectangle** a défini une méthode supplémentaire appelée **Move**, qui ne fait pas partie de l'interface **IVisual**.

Vous pouvez créer un objet **Rectangle**, *r*, et utiliser ses méthodes **Move** et **Paint**, comme vous le souhaitez. Vous pouvez même référencer cet objet à l'aide d'une variable **IVisual**, *v*. Toutefois, en dépit du fait que *v* et *r* référencent le même objet dans la mémoire, vous ne pouvez pas appeler la méthode **Move** à l'aide de la variable *v*, car elle ne fait pas partie de l'interface **IVisual**. Le code suivant donne des exemples :

```
Rectangle r = new Rectangle( );  
r.Move( );           // Ok  
r.Paint( );          // Ok  
IVisual v = (IVisual) r;  
v.Move( );           // Non valide  
v.Paint( );          // Ok
```

Boxing et unboxing



Le langage C# peut convertir des types valeur en références d'objets, et vice-versa.

Système de types unifiés

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

Les valeurs de types tels que **int** ou **bool** peuvent donc être traitées en tant que valeurs simples dans la plupart des cas. Il s'agit généralement de la technique la plus efficace, car aucune surcharge n'est associée aux références. Toutefois, lorsque vous souhaitez utiliser ces valeurs comme s'il s'agissait de références, elles peuvent être temporairement converties (*boxed*) dans ce but.

Boxing

Les expressions de types valeur peuvent également être converties en valeurs de type **object**, puis reconverties en sens inverse. Lorsqu'une variable de type valeur doit être convertie en type **object**, un objet *box* est alloué pour contenir la valeur, et cette dernière est copiée dans l'objet box. Ce processus est appelé *boxing*.

```
int p = 123;
object box;
box = p;           // Boxing (implicite)
box = (object) p;  // Boxing (explicite)
```

L'opération de boxing peut être effectuée de manière implicite, ou de manière explicite en effectuant un cast d'un objet. Le boxing se produit généralement lorsqu'un type valeur est passé dans un paramètre de type **object**.

Unboxing

Lorsqu'une valeur dans un objet est convertie à nouveau en un type valeur, la valeur est copiée de l'objet box dans l'emplacement de stockage approprié. Ce processus est appelé *unboxing*.

```
p = (int) box;    // Unboxing
```

Vous pouvez effectuer l'unboxing avec un opérateur de cast explicite.

Si la valeur dans la référence n'est pas exactement du même type que le cast, ce dernier lève une exception **InvalidCastException**.

Appel de méthodes Object sur des types valeur

Comme le boxing peut être effectué de manière implicite, vous pouvez appeler des méthodes de type object sur toute variable ou expression, même celles qui sont associées à des types valeur. Vous trouverez ci-dessous un exemple :

```
static void Show(object o)
{
    Console.WriteLine(o.ToString( ));
}
Show(42);
```

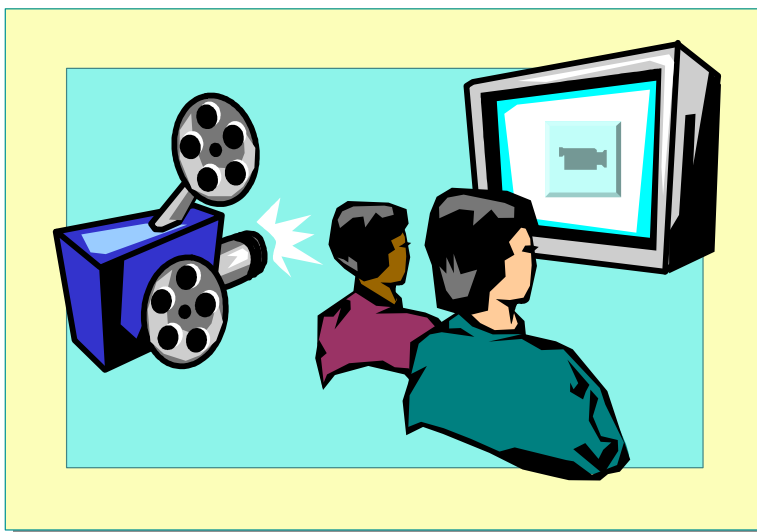
Cela fonctionne parce que la valeur 42 est « boxée » de manière implicite dans un paramètre **object**, et la méthode **ToString** de ce paramètre est ensuite appelée.

Cette opération produit le même résultat que le code suivant :

```
object o = (object) 42; // Box
Console.WriteLine(o.ToString( ));
```

Remarque Le boxing n'a *pas* lieu lorsque vous appelez les méthodes **Object** *directement* sur une valeur. Par exemple, l'expression `42.ToString()` ne place pas 42 dans un type **object** par boxing. Cela vient du fait que le compilateur peut déterminer de manière statique le type et la méthode à appeler.

Présentation multimédia : Casting de type sécurisé en C#



Atelier 8.2 : Conversion de données



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- convertir des valeurs entre un type référence et un autre ;
- tester si une variable référence prend en charge une interface donnée.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les éléments suivants :

- concepts de la programmation orientée objet ;
- création de classes ;
- définition de méthodes.

Durée approximative de cet atelier : 30 minutes

Exercice 1

Test de l'implémentation d'une interface

Dans cet exercice, vous allez ajouter une méthode statique appelée **IsItFormattable** à la classe **Utils** que vous avez créée dans l'atelier 5. Si vous n'avez pas terminé cet atelier, vous pouvez obtenir une copie de cette classe dans le dossier *dossier d'installation*\Labs\Lab08\Starter.

La méthode **IsItFormattable** prend un paramètre de type **object** et teste si ce paramètre implémente l'interface **System.IFormattable**. Si l'objet a effectivement cette interface, la méthode retourne **true**. Dans le cas contraire, elle retourne **false**.

Une classe implémente l'interface **System.IFormattable** pour retourner une représentation de type string d'une instance de cette classe. Les types de base tels que **int** et **ulong** implémentent cette interface (après le boxing de la valeur). De nombreux types référence, tels que **string**, ne le font pas. Les types définis par l'utilisateur peuvent implémenter l'interface si le développeur le demande. Pour plus d'informations sur cette interface, consultez la documentation de l'aide du Kit de développement .NET Framework SDK.

Vous allez écrire un code de test qui appellera la méthode **Utils.IsItFormattable** avec des arguments de différents types et affichera les résultats à l'écran.

► Pour créer la méthode **IsItFormattable**

1. Ouvrez le projet *InterfaceTest.sln* situé dans le dossier *dossier d'installation*\Labs\Lab08\Starter\InterfaceTest.
2. Modifiez la classe **Utils** comme suit :
 - a. Créez une méthode statique publique appelée **IsItFormattable** dans la classe **Utils**.
 - b. Cette méthode prend un paramètre appelé *x* de type **object** qui est passé par valeur. La méthode renvoie une valeur **bool**.
 - c. Utilisez l'opérateur **is** pour déterminer si l'objet passé prend en charge l'interface **System.IFormattable**. Si tel est le cas, la méthode renvoie **true** ; dans le cas contraire, elle renvoie **false**.

La méthode doit ressembler à ce qui suit :

```
using System;

...

class Utils
{
    public static bool IsItFormattable(object x)
    {
        // Utilise l'opérateur is pour tester si
        // l'objet a l'interface IFormattable

        if (x is IFormattable)
            return true;
        else
            return false;
    }
}
```

► **Pour tester la méthode IsItFormattable**

1. Modifiez la classe du fichier **Test**.
2. Dans la méthode **Main**, déclarez et initialisez des variables de types **int**, **ulong** et **string**.
3. Passez chaque variable dans **Utils.IsItFormattable()** et affichez le résultat de chaque appel.
4. La classe **Test** peut ressembler à ce qui suit :

```
using System;
public class Test
{
    static void Main( )
    {
        int i = 0;
        ulong ul = 0;
        string s = "Test";

        Console.WriteLine("int : {0}",
            ↪      Utils.IsItFormattable(i));
        Console.WriteLine("ulong : {0}",
            ↪      Utils.IsItFormattable(ul));
        Console.WriteLine("String : {0}",
            ↪      Utils.IsItFormattable(s));
    }
}
```

5. Compilez et testez le code. **true** doit s'afficher pour les valeurs **int** et **ulong**, et **false** pour la valeur **string**.

Exercice 2

Utilisation des interfaces

Dans cet exercice, vous allez écrire une méthode **Display** qui utilisera l'opérateur **as** pour déterminer si l'objet passé en tant que paramètre prend en charge une interface définie par l'utilisateur appelée **IPrintable** et qui appellera une méthode de cette interface si cette dernière est prise en charge.

► Pour créer la méthode **Display**

1. Ouvrez le projet `TestDisplay.sln` situé dans le dossier *dossier d'installation*\Labs\Lab08\Starter\TestDisplay.

Le code de départ inclut la définition d'une interface appelée **IPrintable**, qui contient une méthode nommée **Print**. Une classe qui implémente cette interface doit utiliser la méthode **Print** pour afficher sur la console les valeurs contenues dans l'objet. Le fichier du code de départ définit également une classe appelée **Coordinate** qui implémente l'interface **IPrintable**.

Un objet **Coordinate** contient une paire de nombres pouvant définir une position dans un espace bidimensionnel. Il n'est pas nécessaire de comprendre comment fonctionne la classe **Coordinate** (même si cela peut vous intéresser). Tout ce que vous devez savoir est que cette classe implémente l'interface **IPrintable** et que vous pouvez utiliser la méthode **Print** pour afficher son contenu.

2. Modifiez la classe **Utils** comme suit :
 - a. Ajoutez une méthode **void** statique publique appelée **Display** dans la classe **Utils**. Cette méthode doit prendre un paramètre, un type **object** passé par valeur, appelé *item*.
 - b. Dans la méthode **Display**, déclarez une variable d'interface appelée *ip* de type **IPrintable**.
 - c. Convertissez la référence dans le paramètre *item* en une référence à l'interface **IPrintable** qui utilise l'opérateur **as**. Stockez le résultat dans la variable *ip*.
 - d. Si la valeur de *ip* n'est pas **null**, utilisez l'interface **IPrintable** pour appeler la méthode **Print**. Si cette valeur est **null**, l'objet ne prend pas en charge l'interface. Dans ce cas, utilisez **Console.WriteLine** pour afficher les résultats de la méthode **ToString** dans le paramètre.

La méthode doit ressembler à ce qui suit :

```
public static void Display(object item)
{
    IPrintable ip;

    ip = (item as IPrintable);

    if (ip != null)
        ip.Print( );
    else
        Console.WriteLine(item.ToString( ));
}
```

► Pour tester la méthode Display

1. Dans la méthode **Main** de la classe **Test**, créez une variable de type **int**, une variable de type **string** et une variable de type **Coordinate**. Pour initialiser la variable **Coordinate**, vous pouvez utiliser le constructeur à deux paramètres :
`Coordinate c = new Coordinate(21.0, 68.0);`

2. Passez ces trois variables, à tour de rôle, dans **Utils.Display** pour les afficher.
3. Le code doit ressembler à ce qui suit :

```
public class Test
{
    static void Main( )
    {
        int    num = 65;
        string msg = "Une chaîne";
        Coordinate c = new Coordinate(21.0,68.0);

        Utils.Display(num);
        Utils.Display(msg);
        Utils.Display(c);
    }
}
```

4. Compilez et testez votre application.

Contrôle des acquis

- Utilisation des variables de type référence
- Utilisation des types référence courants
- Hiérarchie des objets
- Espaces de noms du .NET Framework
- Conversions de données

1. Expliquez comment la mémoire est allouée et désallouée pour une variable de type référence.
2. Quelle valeur spéciale indique qu'une variable référence ne contient pas la référence à un objet ? Que se passe-t-il si vous tentez d'accéder à une variable référence avec cette valeur ?
3. Citez les principales fonctionnalités de la classe **String**.
4. Quel est le type de base de toutes les classes ?

5. Expliquez la différence entre l'opérateur de cast et l'opérateur **as** lors de la conversion entre des références de classes.
6. Citez les différentes manières de déterminer le type d'un objet.

Module 9 : Création et destruction d'objets

Table des matières

Vue d'ensemble	1
Utilisation de constructeurs	2
Initialisation des données	14
Atelier 9.1 : Création d'objets	34
Objets et mémoire	42
Gestion des ressources	48
Atelier 9.2 : Gestion des ressources	57
Contrôle des acquis	60



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Utilisation de constructeurs
- Initialisation des données
- Objets et mémoire
- Gestion des ressources

Dans ce module, vous allez apprendre ce qui se passe lors de la création d'un objet, comment utiliser des constructeurs pour initialiser des objets et comment utiliser des destructeurs pour détruire ces derniers. Vous allez également découvrir ce qui se passe lors de la destruction d'un objet et comment le garbage collection récupère la mémoire.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- utiliser des constructeurs pour initialiser des objets ;
- créer des constructeurs surchargés acceptant des paramètres qui varient ;
- décrire la durée de vie d'un objet et ce qu'il se passe lors de sa destruction ;
- créer des destructeurs ;
- implémenter la méthode **Dispose**.

◆ Utilisation de constructeurs

- Création d'objets
- Utilisation du constructeur par défaut
- Substitution du constructeur par défaut
- Surcharge des constructeurs

Les constructeurs sont des méthodes spéciales que vous utilisez pour initialiser des objets lorsque vous les créez. Même si vous n'écrivez pas vous-même un constructeur, un constructeur par défaut vous est proposé chaque fois que vous créez un objet à partir d'un type référence.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- utiliser les constructeurs par défaut ;
- utiliser les constructeurs pour contrôler ce qui se passe lors de la création d'un objet.

Création d'objets

■ Étape 1 : allocation de mémoire

- Utilisez le mot clé **new** pour allouer de la mémoire à partir du tas

■ Étape 2 : initialisation de l'objet à l'aide d'un constructeur

- Utilisez le nom de la classe suivi de parenthèses

```
Date when = new Date( );
```

Le processus de création d'un objet en C# est constitué de deux étapes :

1. Utilisation du mot clé **new** pour acquérir de la mémoire et l'allouer pour l'objet.
2. Écriture d'un constructeur pour transformer la mémoire acquise par le mot clé **new** en objet.

Même si ce processus est constitué de deux étapes, vous devez les rassembler en une seule expression. Par exemple, si **Date** est le nom d'une classe, utilisez la syntaxe suivante pour allouer de la mémoire et initialiser l'objet **when** :

```
Date when = new Date( );
```

Étape 1 : allocation de mémoire

La première étape de la création d'un objet consiste à allouer de la mémoire pour l'objet. Tous les objets sont créés à l'aide de l'opérateur **new**. Il n'existe aucune exception à cette règle. Vous pouvez procéder explicitement dans votre code ou laisser le compilateur s'en charger.

Le tableau suivant présente des exemples de code et ce qu'ils représentent.

Exemple de code	Représente
<code>string s = "Hello";</code>	<code>string s = new string(new char[] { 'H', 'e', 'l', 'l', 'o' });</code>
<code>int[] array = { 1, 2, 3, 4 };</code>	<code>int[] array = new int[4] { 1, 2, 3, 4 };</code>

Étape 2 : initialisation de l'objet à l'aide d'un constructeur

La seconde étape de création d'un objet consiste à appeler un constructeur. Un constructeur transforme la mémoire allouée par le mot clé **new** en objet. Il existe deux types de constructeurs : les constructeurs d'instances et les constructeurs statiques. Les constructeurs d'instances initialisent les objets. Les constructeurs statiques initialisent les classes.

Collaboration entre le mot clé **new** et les constructeurs d'instances

Il est important de comprendre l'étroitesse de la relation entre le mot clé **new** et les constructeurs d'instances pour la création d'objets. Le seul objectif du mot clé **new** est d'acquérir de la mémoire brute non initialisée. Le seul objectif d'un constructeur d'instance est d'initialiser la mémoire et de la convertir en un objet prêt à être utilisé. Plus particulièrement, **new** n'est impliqué en aucune manière dans l'initialisation et les constructeurs d'instances ne sont impliqués en aucune manière dans l'acquisition de mémoire.

Même si le mot clé **new** et les constructeurs d'instances effectuent des tâches séparées, en tant que programmeur vous ne pouvez pas les utiliser séparément. C'est l'un des moyens dont dispose C# pour garantir qu'une valeur en mémoire est toujours valide avant sa lecture. (C'est ce que l'on appelle une *assignment définitive*.)

Remarque destinée aux programmeurs C++ En C++, vous pouvez allouer de la mémoire sans l'initialiser (en appelant directement l'opérateur **new**). Vous pouvez également initialiser de la mémoire allouée précédemment (en utilisant le mot clé de placement **new**). Cette séparation est impossible en C#.

Utilisation du constructeur par défaut

■ Fonctionnalités d'un constructeur par défaut

- Accessibilité publique
- Même nom que celui de la classe
- Aucun type de retour, pas même **void**
- N'attend aucun argument
- Initialise tous les champs à **zéro, false** ou **null**

■ Syntaxe du constructeur

```
class Date { public Date( ) { ..... } }
```

Lorsque vous créez un objet, le compilateur C# fournit un constructeur par défaut si vous n'en écrivez pas un vous-même. Examinez l'exemple suivant :

```
class Date
{
    private int ccyy, mm, dd;
}

class Test
{
    static void Main( )
    {
        Date when = new Date( );
        ...
    }
}
```

L'instruction dans **Test.Main** crée un objet **Date** appelé **when** à l'aide de l'opérateur **new** (qui alloue la mémoire à partir du tas) et en appelant une méthode spéciale portant le même nom que la classe (le constructeur d'instance). Cependant, la classe **Date** ne déclare pas de constructeur d'instance. (Elle ne déclare aucune méthode). Par défaut, le compilateur crée automatiquement un constructeur d'instance par défaut.

Fonctionnalités d'un constructeur par défaut

Conceptuellement, le constructeur d'instance que le compilateur génère pour la classe **Date** ressemble à l'exemple suivant :

```
class Date
{
    public Date( )
    {
        ccyy = 0;
        mm = 0;
        dd = 0;
    }
    private int ccyy, mm, dd;
}
```

Ce constructeur est doté des fonctionnalités suivantes :

- Même nom que celui de la classe

Par définition, un constructeur d'instance est une méthode portant le même nom que sa classe. Il s'agit d'une définition naturelle et intuitive qui correspond à la syntaxe que vous avez déjà vue. En voici un exemple :

```
Date when = new Date( );
```

- Aucun type de retour

Il s'agit de la deuxième caractéristique d'un constructeur : il ne possède jamais de type de retour, pas même **void**.

- Aucun argument requis

Il est possible de déclarer des constructeurs prenant des arguments. Cependant, le constructeur par défaut généré par le compilateur n'attend aucun argument.

- Tous les champs sont initialisés à zéro

Ce point est important. Le constructeur par défaut généré par le compilateur initialise implicitement tous les champs non statiques comme suit :

- les champs numériques (tels que **int**, **double** et **decimal**) sont initialisés à zéro ;
- les champs de type **bool** sont initialisés à **false** ;
- les types référence (que nous avons vus dans un module précédent) sont initialisés à **null** ;
- les champs de type **struct** sont initialisés pour contenir des valeurs zéro dans tous leurs éléments.

- Accessibilité publique

Elle permet la création d'instances de l'objet.

Remarque Le module 10, « Héritage dans C# », du cours 2132A, *Programmation en C#*, aborde le sujet des classes abstraites. Le constructeur par défaut créé par le compilateur pour une classe abstraite a un accès protégé.

Substitution du constructeur par défaut

- Il se peut que le constructeur par défaut soit inadéquat

- S'il tel est le cas, ne l'utilisez pas et écrivez le vôtre !

```
class Date
{
    public Date( )
    {
        ccy = 1970;
        mm = 1;
        dd = 1;
    }
    private int ccy, mm, dd;
}
```

Quelquefois il ne convient pas d'utiliser le constructeur par défaut créé par le compilateur. Vous pouvez alors écrire votre propre constructeur contenant seulement le code destiné à initialiser les champs à des valeurs autres que zéro. Tous les champs que vous n'initialisez pas dans votre constructeur conservent l'initialisation par défaut à zéro.

Que se passe-t-il si le constructeur par défaut est inadéquat ?

Il existe plusieurs cas de figures dans lesquels le constructeur par défaut créé par le compilateur peut s'avérer inadéquat :

- L'accès public est quelquefois inadéquat.

Le modèle de fabrication (ou modèle *Factory Method*) utilise un constructeur qui n'est pas public. (Ce modèle est abordé dans *Design Patterns : Catalogue de modèles de conception réutilisables* de E. Gamma, R. Helm, R. Johnson et J. Vlissides. Nous le verrons dans un module ultérieur.)

Les fonctions procédurales (telles que **Cos** et **Sin**) font souvent appel à des constructeurs privés.

Le modèle de Singleton utilise généralement un constructeur privé. (Le modèle de Singleton est également abordé dans *Design Patterns : Catalogue de modèles de conception réutilisables* et dans une prochaine rubrique de cette section.)

- L'initialisation à zéro est quelquefois inadéquate.

Observez le constructeur par défaut créé par le compilateur pour la classe **Date** suivante :

```
class Date
{
    private int ccyy, mm, dd;
}
```

Ce constructeur initialisera les champs année (*ccyy*), mois (*mm*) et jour (*dd*) à zéro. Cela ne sera pas adéquat si vous voulez que la valeur par défaut de la date soit différente.

- Le code invisible est difficile à gérer.

En effet, vous ne voyez pas le code du constructeur par défaut, ce qui peut poser un problème. Vous ne pouvez pas par exemple exécuter du code invisible pas à pas lors du débogage. Par ailleurs, si vous décidez d'utiliser l'initialisation par défaut à zéro, comment les développeurs souhaitant mettre à jour le code sauront-ils qu'il s'agissait d'un choix délibéré ?

Écriture de votre propre constructeur par défaut

Si le constructeur par défaut créé par le compilateur est inadéquat, vous devez écrire le vôtre. Le langage C# vous aide dans cette tâche.

Vous pouvez écrire un constructeur contenant seulement le code destiné à initialiser les champs à des valeurs autres que zéro. Tous les champs que vous n'initialisez pas dans votre constructeur conservent l'initialisation par défaut à zéro. Voici un exemple de code :

```
class DefaultInit
{
    public int a, b;
    public DefaultInit( )
    {
        a = 42;
        // b conserve l'initialisation à zéro par défaut
    }
}

class Test
{
    static void Main( )
    {
        DefaultInit di = new DefaultInit( );
        Console.WriteLine(di.a); // Écrit 42
        Console.WriteLine(di.b); // Écrit zéro
    }
}
```


Ne soyez pas tenté d'aller au-delà de la simple initialisation de vos constructeurs. En effet, vous devez envisager l'éventualité d'un échec : la seule manière logique de signaler un échec d'initialisation dans un constructeur consiste à lever une exception.

Remarque Il en va de même pour les opérateurs, qui sont présentés dans le module 12, « Opérateurs, délégués et événements », du cours 2132A, *Programmation en C#*.

En cas de réussite de l'initialisation, vous pouvez utiliser votre objet.
En cas d'échec de l'initialisation, vous vous retrouvez sans objet.

Surcharge des constructeurs

- **Les constructeurs sont des méthodes et peuvent être surchargés**
 - Même portée, même nom, paramètres différents
 - Les objets peuvent être initialisés de plusieurs manières
- **AVERTISSEMENT**
 - Si vous écrivez un constructeur pour une classe, le compilateur ne crée pas un constructeur par défaut

```
class Date
{
    public Date( ) { .... }
    public Date(int year, int month, int day) { .... }
    ...
}
```

Les constructeurs sont des types particuliers de méthodes. Vous pouvez surcharger les constructeurs tout comme les méthodes.

En quoi consiste la surcharge ?

« Surcharge » est le terme technique désignant le fait de déclarer deux méthodes ou plus dans la même portée avec le même nom. Voici un exemple de code :

```
class Overload
{
    public void Method( ) { ... }
    public void Method(int x) { ... }
}
class Use
{
    static void Main( )
    {
        Overload o = new Overload( );
        o.Method( );
        o.Method(42);
    }
}
```

Dans cet exemple de code, deux méthodes appelées **Method** sont déclarées dans la portée de la classe **Overload** et toutes deux sont appelées dans **Use.Main**. Il n'existe toutefois aucune ambiguïté, car le nombre et les types d'arguments déterminent quelle méthode est appelée.

Initialisation d'un objet de plusieurs manières

La capacité à initialiser un objet de plusieurs façons fut l'une des principales raisons d'acceptation de la pratique de la surcharge. Les constructeurs étant des types particuliers de méthodes, ils peuvent être surchargés exactement comme des méthodes. Cela signifie que vous pouvez définir diverses manières d'initialiser un objet. Voici un exemple de code :

```
class Overload
{
    public Overload( ) { this.data = -1; }
    public Overload(int x) { this.data = x; }
    private int data;
}

class Use
{
    static void Main( )
    {
        Overload o1 = new Overload( );
        Overload o2 = new Overload(42);
        ...
    }
}
```

L'objet **o1** est créé à l'aide du constructeur qui ne prend pas d'argument et la valeur de la variable d'instance privée *data* est définie à -1. L'objet **o2** est créé à l'aide du constructeur qui prend un seul entier et la valeur de la variable d'instance *data* est définie à 42.

Initialisation des champs à des valeurs autres que celles par défaut

Vous trouverez de nombreux cas dans lesquels les champs ne peuvent pas être initialisés à zéro pour des raisons de logique. Vous pourrez alors écrire votre propre constructeur nécessitant un ou plusieurs paramètres qui seront ensuite utilisés pour initialiser les champs. Prenez l'exemple de la classe **Date** suivante :

```
class Date
{
    public Date(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

Le problème avec ce constructeur est qu'il est facile de se tromper dans l'ordre des arguments. Par exemple :

```
Date birthday = new Date(23, 11, 1968); // Erreur
```

Le code devrait être `new Date(1968,11,23)`. Cette erreur ne sera pas identifiée comme une erreur de compilation car les trois arguments sont des entiers. L'une des solutions consiste à utiliser le modèle Valeur entière. Vous pourriez alors transformer *Year*, *Month* et *Day* en **struct** plutôt qu'en valeurs **int**, comme suit :

```
struct Year
{
    public readonly int value;
    public Year(int value) { this.value = value; }
}

struct Month // Ou un enum
{
    public readonly int value;
    public Month(int value) { this.value = value; }
}
struct Day
{
    public readonly int value;
    public Day(int value) { this.value = value; }
}
class Date
{
    public Date(Year y, Month m, Day d)
    {
        ccyy = y.value;
        mm = m.value;
        dd = d.value;
    }
    private int ccyy, mm, dd;
}
```

Conseil L'utilisation de **structs** ou d'**enums** plutôt que de classes pour *Day*, *Month* et *Year* réduit la surcharge lors de la création de l'objet **Date**. Nous verrons cela plus loin dans ce module.

Le code suivant montre un simple changement qui, non seulement capture d'éventuelles erreurs dans l'ordre des arguments, mais vous permet aussi de créer des constructeurs **Date** surchargés aux formats français, américain et ISO :

```
class Date
{
    public Date(Year y, Month m, Day d) { ... } // ISO
    public Date(Month m, Day d, Year y) { ... } // É.-U.
    public Date(Day d, Month m, Year y) { ... } // France
    ...
    private int ccyy, mm, dd;
}
```

Surcharge et constructeur par défaut

Si vous déclarez une classe avec un constructeur, le compilateur ne génère pas le constructeur par défaut. Dans l'exemple suivant, la classe **Date** étant déclarée avec un constructeur, l'expression `new Date()` ne sera pas compilée :

```
class Date
{
    public Date(Year y, Month m, Day d) { ... }
    // Aucun autre constructeur
    private int ccyy, mm, dd;
}
class Fails
{
    static void Main( )
    {
        Date defaulted = new Date( ); // Erreur de compilation
    }
}
```

Cela signifie que si vous voulez pouvoir créer des objets **Date** sans fournir d'arguments de constructeur, vous devrez déclarer explicitement un constructeur surchargé par défaut, comme dans l'exemple suivant :

```
class Date
{
    public Date( ) { ... }
    public Date(Year y, Month m, Day d) { ... }
    ...
    private int ccyy, mm, dd;
}
class Succeeds
{
    static void Main( )
    {
        Date defaulted = new Date( ); // OK
    }
}
```

◆ Initialisation des données

- Utilisation de listes d'initialiseur
- Déclaration de variables readonly et de constantes
- Initialisation des champs readonly
- Déclaration d'un constructeur pour un struct
- Utilisation de constructeurs privés
- Utilisation de constructeurs statiques

Vous avez vu les éléments de base des constructeurs. Mais ces derniers ont d'autres fonctionnalités et utilisations.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- initialiser les données dans des objets à l'aide de constructeurs ;
- utiliser des constructeurs privés ;
- utiliser des constructeurs statiques.

Utilisation de listes d'initialiseur

- **Les constructeurs surchargés peuvent contenir du code dupliqué**

- Restructurez-les en obligeant les constructeurs à s'appeler entre eux
- Utilisez le mot clé **this** dans une liste d'initialiseurs

```
class Date
{
    ...
    public Date( ) : this(1970, 1, 1) { } }
    public Date(int year, int month, int day) { .... }
}
```

Vous pouvez avoir recours à une syntaxe particulière appelée liste d'initialiseurs pour implémenter un constructeur en appelant un constructeur surchargé.

Prévention des initialisations dupliquées

Le code suivant montre un exemple de constructeurs surchargés avec du code d'initialisation dupliqué :

```
class Date
{
    public Date( )
    {
        ccyy = 1970;
        mm = 1;
        dd = 1;
    }
    public Date(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

Notez la duplication de *dd*, *mm* et *ccyy* à gauche des trois initialisations. Il ne s'agit pas d'une duplication extensive, mais elle n'en reste pas moins une duplication, que vous devez éviter dans la mesure du possible. Par exemple, supposons que vous décidiez de changer la représentation d'une **Date** en un champ **long**. Vous devriez alors ré-écrire tous les constructeurs **Date**.

Structuration/restructuration (factoring/refactoring) des initialisations dupliquées

Une méthode standard de restructuration du code dupliqué consiste à extraire le code commun dans sa propre méthode. Voici un exemple de code :

```
class Date
{
    public Date( )
    {
        Init(1970, 1, 1);
    }
    public Date(int year, int month, int day)
    {
        Init(day, month, year);
    }
    private void Init(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

Cette solution est meilleure que la précédente. Si vous changez la représentation d'une **Date** en champ **long**, vous avez simplement besoin de modifier **Init**. Malheureusement, ce type de restructuration des constructeurs fonctionne quelquefois mais pas systématiquement. Par exemple, il ne fonctionne pas si vous tentez de restructurer l'initialisation d'un champ **readonly**. (Nous verrons cela plus loin dans ce module). Les langages de programmation orientés objet fournissent des mécanismes permettant de résoudre ce problème connu. Par exemple, en C++ vous pouvez adopter des valeurs par défaut. En C# vous utilisez des listes d'initialiseurs.

Utilisation d'une liste d'initialiseurs

Une liste d'initialiseurs vous permet d'écrire un constructeur appelant un autre constructeur dans la même classe. Vous écrivez la liste d'initialiseurs entre la parenthèse de fermeture et le crochet d'ouverture du constructeur. Une liste d'initialiseurs commence par le signe deux-points, est suivie du mot clé **this**, puis d'arguments entre parenthèses. Par exemple, dans le code suivant, le constructeur par défaut **Date** (celui qui ne possède pas d'arguments) utilise une liste d'initialiseurs pour appeler le second constructeur **Date** avec trois arguments : 1970, 1 et 1.

```
class Date
{
    public Date( ) : this(1970, 1, 1)
    {
    }
    public Date(int year, int month, int day)
    {
        ccyy = year;
        mm = month;
        dd = day;
    }
    private int ccyy, mm, dd;
}
```

Cette syntaxe est efficace, elle fonctionne toujours et si vous l'utilisez, vous n'avez pas besoin de créer une autre méthode **Init**.

Restrictions liées aux listes d'initialiseurs

Vous devez respecter trois restrictions lors de l'initialisation de constructeurs :

- Vous ne pouvez utiliser les listes d'initialiseurs dans des constructeurs que comme dans l'exemple suivant :

```
class Point
{
    public Point(int x, int y) { ... }
    // Erreur de compilation
    public void Init( ) : this(0, 0) { }
}
```

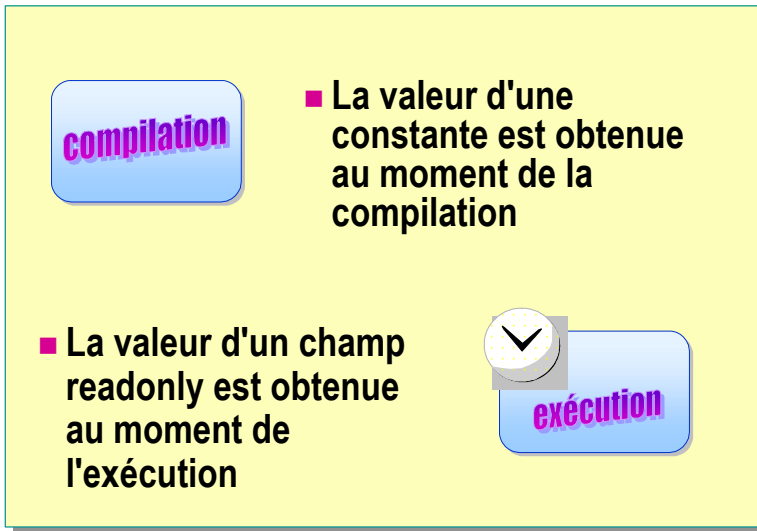
- Vous ne pouvez pas écrire une liste d'initialiseurs qui s'appelle elle-même. Voici un exemple de code :

```
class Point
{
    // Erreur de compilation
    public Point(int x, int y) : this(x, y) { }
}
```

- Vous ne pouvez pas utiliser le mot clé **this** dans une expression destinée à créer un argument de constructeur. Voici un exemple de code :

```
class Point
{
    // Erreur de compilation
    public Point( ) : this(X(this), Y(this)) { }
    public Point(int x, int y) { ... }
    private static int X(Point p) { ... }
    private static int X(Point p) { ... }
}
```

Déclaration de variables readonly et de constantes



Lorsque vous utilisez des constructeurs, vous avez besoin de savoir comment déclarer des variables **readonly** et des constantes.

Utilisation de variables Readonly

Vous pouvez qualifier un champ comme étant **readonly** dans sa déclaration, comme suit :

```
readonly int nLoopCount = 10;
```

Vous obtiendrez une erreur de compilation si vous tentez de modifier cette valeur.

Utilisation de variables constantes

Une variable constante représente une valeur constante calculée au moment de la compilation. À l'aide des variables constantes, vous pouvez définir des variables dont la valeur ne change jamais, comme dans l'exemple suivant :

```
const int speedLimit = 55;
```

Les constantes peuvent dépendre d'autres constantes dans le même programme tant que les dépendances ne sont pas de nature circulaire. Le compilateur évalue automatiquement les déclarations de constantes dans l'ordre approprié.

Initialisation des champs readonly

■ Les champs readonly doivent être initialisés

- Implicitement à zéro, **false** ou **null**
- Explicitement à leur déclaration dans un initialiseur de variable
- Explicitement dans un constructeur d'instance

```
class SourceFile
{
    private readonly ArrayList lines;
}
```

Les champs ne pouvant être réassignés et devant être initialisés sont des champs **readonly**. Pour initialiser un champ **readonly**, trois possibilités s'offrent à vous :

- utiliser l'initialisation par défaut d'un champ **readonly** ;
- initialiser un champ **readonly** dans un constructeur ;
- initialiser des champs **readonly** à l'aide d'un initialiseur de variables.

Utilisation de l'initialisation par défaut d'un champ readonly

Le constructeur par défaut créé par le compilateur initialise tous les champs (qu'ils soient **readonly** ou non) à leur valeur par défaut, qu'elle soit zéro, **false** ou **null**. Voici un exemple de code :

```
class SourceFile
{
    public readonly ArrayList lines;
}
class Test
{
    static void Main( )
    {
        SourceFile src = new SourceFile( );
        Console.WriteLine(src.lines == null); // True
    }
}
```

Le constructeur **SourceFile** n'existant pas, le compilateur écrit un constructeur par défaut, qui initialise le champ *lines* à **null**. En conséquence, l'instruction **WriteLine** de l'exemple ci-dessus écrit « *True* ».

Si vous déclarez votre propre constructeur dans une classe sans initialiser explicitement un champ **readonly**, le compilateur initialise quand même le champ. En voici un exemple :

```
class SourceFile
{
    public SourceFile( ) { }
    public readonly ArrayList lines;
}
class Test
{
    static void Main( )
    {
        SourceFile src = new SourceFile( );
        Console.WriteLine(src.lines == null); // Toujours true
    }
}
```

Cela n'est pas très utile. En effet, dans le cas présent, le champ **readonly** est initialisé à **null** et reste **null** car il est impossible de réassigner un champ **readonly**.

Initialisation d'un champ **readonly** dans un constructeur

Vous pouvez initialiser explicitement un champ **readonly** dans le corps d'un constructeur. En voici un exemple :

```
class SourceFile
{
    public SourceFile( )
    {
        lines = new ArrayList( );
    }
    private readonly ArrayList lines;
}
```

L'instruction au sein du constructeur ressemble dans sa syntaxe à une assignation à *lines*, qui ne serait normalement pas autorisée car *lines* est un champ **readonly**. Cependant, l'instruction est compilée car le compilateur reconnaît que l'assignation survient dans le corps d'un constructeur et la traite donc comme une initialisation.

L'un des avantages de l'initialisation de champs **readonly** de cette façon est que vous pouvez faire appel à des paramètres de constructeurs dans l'expression **new**. En voici un exemple :

```
class SourceFile
{
    public SourceFile(int suggestedSize)
    {
        lines = new ArrayList(suggestedSize);
    }
    private readonly ArrayList lines;
}
```

Initialisation des champs readonly à l'aide d'un initialiseur de variables

Vous pouvez initialiser un champ **readonly** directement à sa déclaration à l'aide d'un initialiseur de variables. En voici un exemple :

```
class SourceFile
{
    public SourceFile( )
    {
        ...
    }
    private readonly ArrayList lines = new ArrayList( );
}
```

Il s'agit simplement d'une abréviation pratique. Le compilateur réécrit conceptuellement une initialisation de variable (qu'elle soit **readonly** ou non) dans une assignation au sein de tous les constructeurs. Par exemple, la classe ci-dessus sera convertie conceptuellement dans la classe suivante :

```
class SourceFile
{
    public SourceFile( )
    {
        lines = new ArrayList( );
        ...
    }
    private readonly ArrayList lines;
}
```

Déclaration d'un constructeur pour un struct

■ Le compilateur

- Génère toujours un constructeur par défaut. Les constructeurs par défaut initialisent automatiquement tous les champs à zéro.

■ Le programmeur

- Peut déclarer les constructeurs avec un ou plusieurs arguments. Les constructeurs déclarés n'initialisent pas automatiquement tous les champs à zéro.
- Ne peut pas déclarer de constructeur par défaut.
- Ne peut pas déclarer de constructeur protégé.

La syntaxe utilisée pour déclarer un constructeur est identique dans le cas d'un struct à celle utilisée pour une classe. Voici par exemple un **struct** appelé **Point** doté d'un constructeur :

```
struct Point
{
    public Point(int x, int y) { ... }
    ...
}
```

Restrictions liées aux constructeurs de struct

Même si la syntaxe des constructeurs de struct et de classe est la même, certaines restrictions ne s'appliquent qu'aux constructeurs de **struct** :

- Le compilateur crée toujours un constructeur de **struct** par défaut.
- Vous ne pouvez pas déclarer un constructeur par défaut dans un **struct**.
- Vous ne pouvez pas déclarer un constructeur protégé dans un **struct**.
- Vous devez initialiser tous les champs.

Le compilateur crée toujours un constructeur de struct par défaut

Le compilateur génère toujours un constructeur par défaut, que vous déclariez ou non les constructeurs vous-même (à l'inverse des classes, dans lesquelles le compilateur génère le constructeur par défaut uniquement si vous ne déclarez aucun constructeur vous-même). Le constructeur de **struct** généré par le compilateur initialise tous les champs à zéro, **false** ou **null**.

```
struct SPoint
{
    public SPoint(int x, int y) { ... }
    ...
    static void Main( )
    {
        // OK
        SPoint p = new SPoint( );
    }
}
class CPoint
{
    public CPoint(int x, int y) { ... }
    ...
    static void Main( )
    {
        // Erreur de compilation
        CPoint p = new CPoint( );
    }
}
```

Cela signifie qu'une valeur de struct créée à l'aide de

```
SPoint p = new SPoint( );
```

crée une valeur de struct sur la pile (l'utilisation de l'opérateur **new** pour créer un struct n'acquiert pas de mémoire à partir du tas) et initialise les champs à zéro. Il est impossible de changer ce comportement.

Cependant, une valeur de struct créée à l'aide de

```
SPoint p;
```

crée toujours une valeur de struct sur la pile, mais elle n'initialise aucun des champs (tous les champs doivent donc être définitivement assignés avant d'être référencés). Vous trouverez ci-dessous un exemple :

```
struct SPoint
{
    public int x, y;
    ...
    static void Main( )
    {
        SPoint p1;
        Console.WriteLine(p1.x); // Erreur de compilation
        SPoint p2;
        p2.x = 0;
        Console.WriteLine(p2.x); // OK
    }
}
```

Conseil Veillez à ce que tous les types **struct** que vous définissez soient valides et que tous les champs soient définis à zéro.

Vous ne pouvez pas déclarer un constructeur par défaut dans un struct

En effet, étant donné que le compilateur crée toujours un constructeur par défaut dans un struct (comme nous venons de le décrire), vous finiriez par vous retrouver avec une définition dupliquée.

```
class CPoint
{
    // OK, car CPoint est une classe
    public CPoint( ) { ... }
    ...
}
struct SPoint
{
    // Erreur de compilation car SPoint est un struct
    public SPoint( ) { ... }
    ...
}
```

Vous pouvez déclarer un constructeur de struct tant qu'il attend au moins un argument. Si vous déclarez un constructeur de struct, il n'initialise pas automatiquement les champs à une valeur par défaut (à la différence d'un constructeur de struct par défaut généré par le compilateur).

```
struct SPoint
{
    public SPoint(int x, int y) { ... }
    ...
}
```

Vous ne pouvez pas déclarer un constructeur par défaut dans un struct

En effet, étant donné que vous ne pouvez jamais dériver d'autres classes ou structs d'un struct, une restriction d'accès n'aurait aucun sens, comme le montre l'exemple suivant :

```
class CPoint
{
    // OK
    protected CPoint(int x, int y) { ... }
}
struct SPoint
{
    // Erreur de compilation
    protected SPoint(int x, int y) { ... }
}
```

Vous devez initialiser tous les champs

Si vous déclarez un constructeur de classe qui échoue lors de l'initialisation d'un champ, le compilateur s'assure que le champ conserve toutefois son initialisation à zéro par défaut. Voici un exemple de code :

```
class CPoint
{
    private int x, y;
    public CPoint(int x, int y) { /*rien*/ }
    // OK. Le compilateur s'assure que x et y
    // sont initialisés à zéro.
}
```

Cependant, si vous déclarez un constructeur de struct qui échoue lors de l'initialisation d'un champ, le compilateur génère une erreur de compilation :

```
struct SPoint1 // OK : initialisé lors de la déclaration
{
    private int x,y;
    public SPoint1(int a, int b) { }
}
struct SPoint1 // OK : initialisé dans le constructeur
{
    private int x, y;
    public SPoint2(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Utilisation de constructeurs privés

■ Un constructeur privé empêche la création accidentelle d'objets

- Les méthodes d'instance ne peuvent pas être appelées.
- Les méthodes statiques peuvent être appelées.
- Voici une façon pratique d'implémenter les fonctions procédurales

```
public class Math
{
    public static double Cos(double x) { .... }
    public static double Sin(double x) { .... }
    private Math( ) { } }
}
```

Jusqu'à présent, vous avez vu comment utiliser des constructeurs publics. C# propose également des constructeurs privés, qui sont utiles dans certaines applications.

Utilisation de constructeurs privés pour les fonctions procédurales

La programmation orientée objet propose un puissant paradigme pour structurer le logiciel dans de très nombreux domaines. Ce paradigme ne peut cependant pas être appliqué dans tous les cas. Par exemple, le calcul du sinus ou du cosinus d'un nombre à virgule flottante en double précision n'a rien d'une opération orientée objet.

Déclaration de fonctions

La façon la plus intuitive de calculer le sinus ou le cosinus consiste à utiliser des fonctions globales définies en dehors d'un objet, comme suit :

```
double Cos(double x) { ... }
double Sin(double x) { ... }
```

Le code ci-dessus n'est pas autorisé en C#. Les fonctions globales sont possibles dans des langages procéduraux tels que le C et dans les langages hybrides tels que C++, mais ne sont pas autorisées en C#. En C#, vous devez déclarer les fonctions au sein d'une classe ou d'un struct, comme suit :

```
class Math
{
    public double Cos(double x) { ... }
    public double Sin(double x) { ... }
}
```

Déclaration de méthodes statiques et de méthodes d'instance

La technique exposée dans l'exemple précédent pose un problème : **Cos** et **Sin** étant des méthodes d'instance, vous êtes obligé de créer un objet **Math** à partir duquel appeler **Sin** ou **Cos**, comme le montre le code suivant :

```
class Cumbersome
{
    static void Main( )
    {
        Math m = new Math( );
        double answer;
        answer = m.Cos(42.0);
        // Ou
        answer = new Math( ).Cos(42.0);
    }
}
```

Cependant, vous pouvez facilement résoudre ce problème en déclarant **Cos** et **Sin** comme méthodes statiques, comme suit :

```
class Math
{
    public static double Cos(double x) { ... }
    public static double Sin(double x) { ... }
    private Math( ) { ... }
}
class LessCumbersome
{
    static void Main( )
    {
        double answer = Math.Cos(42.0);
    }
}
```

Avantages des méthodes statiques

Si vous déclarez **Cos** en tant que méthode statique, la syntaxe pour l'utiliser devient :

- Plus simple

Vous ne pouvez appeler **Cos** que d'une seule façon (au moyen de **Math**), au lieu de deux dans l'exemple précédent (au moyen de **m** et de **new Math()**).

- Plus rapide

Vous n'avez plus besoin de créer un nouvel objet **Math**.

Il reste toutefois un petit problème. Le compilateur génère un constructeur par défaut doté d'un accès public, qui vous permet de créer des objets **Math**. Ces objets n'ont aucune utilité, car la classe **Math** contient uniquement des méthodes statiques. Vous pouvez empêcher la création d'objets **Math** de deux façons :

- Déclarer **Math** en tant que classe abstraite.

Cette opération est déconseillée, car l'intérêt d'une classe abstraite est de pouvoir en dériver d'autres.

- Déclarer un constructeur **Math** privé.

Cette solution est meilleure. En effet, lorsque vous déclarez un constructeur dans la classe **Math**, vous empêchez le compilateur de générer le constructeur par défaut. Si vous déclarez également le constructeur comme étant privé, vous empêchez la création d'objets **Math**. Le constructeur privé empêche également l'utilisation de **Math** comme classe de base.

Le modèle de Singleton

Le modèle de Singleton (qui est abordé dans *Design Patterns : Catalogue de modèles de conception réutilisables*) a pour objectif de « s'assurer qu'il n'existe qu'une seule instance d'une classe donnée et d'y proposer un point global d'accès ». La technique consistant à déclarer une classe à l'aide d'un constructeur privé et de méthodes statiques est quelquefois suggérée comme moyen d'implémenter le modèle de Singleton.

Remarque L'un des aspects importants du modèle de Singleton est qu'une classe ne possède qu'une seule instance. Avec un constructeur privé et des méthodes statiques, il n'existe aucune instance. L'implémentation canonique du modèle de Singleton consiste à créer une méthode donnant accès à l'instance unique et à utiliser ensuite cette instance pour appeler les méthodes d'instance.

Utilisation de constructeurs statiques

■ Rôle

- Ils sont appelés par le chargeur de classes au moment de l'exécution
- Ils peuvent être utilisés pour initialiser des champs statiques
- Ils sont toujours appelés avant le constructeur d'instance

■ Restrictions

- Ils ne peuvent pas être appelés.
- Ils ne peuvent pas avoir de modificateur d'accès
- Ils ne doivent pas avoir de paramètres

Tout comme un constructeur d'instance garantit qu'un objet se trouve dans un état initial bien défini avant son utilisation, un constructeur statique garantit qu'une classe se trouve dans un état initial bien défini avant son utilisation.

Chargement des classes au moment de l'exécution

C# est un langage dynamique. Lorsque le Common Language Runtime exécute un programme Microsoft® .NET, il trouve souvent du code utilisant une classe qui n'a pas encore été chargée. Dans ce cas, l'exécution est interrompue momentanément, la classe est chargée dynamiquement, puis l'exécution se poursuit.

Initialisation des classes au moment du chargement

C# s'assure qu'une classe est toujours initialisée avant d'être utilisée de quelque manière que ce soit dans le code. Pour ce faire, il utilise des constructeurs statiques.

Vous pouvez déclarer un constructeur statique de la même façon qu'un constructeur d'instance, mais vous devez le faire précéder du mot clé **static**, comme suit :

```
class Example
{
    static Example( ) { ... }
}
```

Une fois que le chargeur de classes a chargé une classe qui sera bientôt utilisée, mais avant qu'il ne poursuive son exécution normale, il exécute le constructeur statique de cette classe. Vous avez ainsi la garantie que les classes sont toujours initialisées avant d'être utilisées. Le minutage exact de l'exécution du constructeur statique dépend de l'implémentation tout en restant soumis aux règles suivantes :

- Le constructeur statique d'une classe est exécuté avant la création de toute instance de la classe.
- Le constructeur statique d'une classe est exécuté avant le référencement de tout membre statique de la classe.
- Le constructeur statique d'une classe s'exécute au plus une fois au cours de l'instanciation d'un programme.

Initialisation des champs statiques et des constructeurs statiques

Un constructeur statique est le plus souvent utilisé pour initialiser les champs statiques d'une classe. En effet, lorsque vous initialisez un champ statique directement à son point de déclaration, le compilateur convertit conceptuellement l'initialisation en une assignation au sein du constructeur statique. En d'autres termes :

```
class Example
{
    private static Wibble w = new Wibble( );
}
```

est converti par le compilateur en

```
class Example
{
    static Example( )
    {
        w = new Wibble( );
    }
    private static Wibble w;
}
```

Restrictions liées aux constructeurs statiques

Comprendre les quatre restrictions suivantes en matière de syntaxe des constructeurs statiques vous permettra de mieux comprendre comment le Common Language Runtime utilise les constructeurs statiques :

- Vous ne pouvez pas appeler un constructeur statique.
- Vous ne pouvez pas déclarer un constructeur statique avec un modificateur d'accès.
- Vous ne pouvez pas déclarer un constructeur statique avec des paramètres.
- Vous ne pouvez pas utiliser le mot clé **this** dans un constructeur statique.

Vous ne pouvez pas appeler un constructeur statique

Tout constructeur statique doit être appelé avant le référencement des instances de la classe dans le code. S'il incombait aux programmeurs d'implémenter cette règle plutôt qu'au runtime du Microsoft .NET Framework, ils risqueraient de ne pas toujours assumer leur responsabilité. Ils pourraient oublier l'appel, voire appeler le constructeur statique plusieurs fois. Le runtime .NET évite ces problèmes potentiels en interdisant les appels de constructeurs statiques dans le code. Seul le runtime .NET peut appeler un constructeur statique.

```
class Point
{
    static Point( ) { ... }
    static void Main( )
    {
        Point.Point( ); // Erreur de compilation
    }
}
```

Vous ne pouvez pas déclarer un constructeur statique avec un modificateur d'accès

Puisque vous ne pouvez pas appeler un constructeur statique, il ne sert à rien d'en déclarer un avec un modificateur d'accès et cela peut entraîner une erreur de compilation :

```
class Point
{
    public static Point( ) { ... } // Erreur de compilation
}
```

Vous ne pouvez pas déclarer un constructeur statique avec des paramètres

Puisque vous ne pouvez pas appeler un constructeur statique, il ne sert à rien d'en déclarer un avec un paramètre et cela peut entraîner une erreur de compilation. Cela signifie également que vous ne pouvez pas déclarer de constructeur statique surchargé. Voici un exemple :

```
class Point
{
    static Point(int x) { ... } // Erreur de compilation
}
```


Vous ne pouvez pas utiliser le mot clé `this` dans un constructeur statique

Puisqu'un constructeur statique initialise la classe et pas les instances d'objets, il n'a pas de référence **this** implicite et, par conséquent, toute tentative d'utilisation du mot clé **this** génère une erreur de compilation :

```
class Point
{
    private int x, y;
    static Point( ) : this(0,0)    // Erreur de compilation
    {
        this.x = 0; // Erreur de compilation
        this.y = 0; // Erreur de compilation
    }
    ...
}
```

Atelier 9.1 : Création d'objets



Objectifs

Dans cet atelier, vous allez modifier la classe **BankAccount** que vous avez créée dans les ateliers précédents de manière à ce qu'elle utilise des constructeurs. Vous créerez également une classe, **BankTransaction**, et vous l'utiliserez pour stocker des informations relatives aux transactions (dépôts et retraits) effectuées sur un compte.

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- substituer le constructeur par défaut ;
- créer des constructeurs surchargés ;
- initialiser des données **readonly**.

Conditions préalables

Avant de poursuivre, vous devez bien connaître les procédures suivantes :

- créer des classes et instancier des objets ;
- définir et appeler des méthodes.

Vous devez également avoir réalisé l'atelier 8.1. Dans le cas contraire, vous pouvez utiliser le code de la solution.

Durée approximative de cet atelier : 60 minutes

Exercice 1

Implémentation de constructeurs

Dans cet exercice, vous allez modifier la classe **BankAccount** que vous avez créée dans les ateliers précédents. Vous allez supprimer les méthodes qui remplissent les variables d'instance du numéro de compte et du type de compte et les remplacer par une série de constructeurs pouvant être utilisés lors de l'instanciation de **BankAccount**.

Vous remplacerez le constructeur par défaut pour générer un numéro de compte (à l'aide de la technique précédemment utilisée), définirez le type de compte à **Courant** et définirez la valeur du solde à zéro.

Vous créerez également trois autres constructeurs prenant différentes combinaisons de paramètres :

- Le premier prendra un **AccountType**. Le constructeur générera un numéro de compte, définira la valeur du solde à zéro et assignera la valeur passée au type du compte.
- Le deuxième prendra un **decimal**. Le constructeur générera un numéro de compte, assignera la valeur du solde à **Courant** et assignera la valeur transmise au solde.
- Le troisième prendra un **AccountType** et un **decimal**. Le constructeur générera un numéro de compte, assignera le type du compte à la valeur du paramètre **AccountType** et assignera la valeur du paramètre **decimal** au solde.

► Pour créer le constructeur par défaut

1. Ouvrez le projet Constructors.sln situé dans le dossier *Fichiers d'atelier\Lab09\Starter\Constructors*.
2. Dans la classe **BankAccount**, supprimez la méthode **Populate**.
3. Créez un constructeur par défaut de la manière suivante :
 - a. Son nom est **BankAccount**.
 - b. Il est public.
 - c. Il ne prend aucun paramètre.
 - d. Il ne possède pas de type de retour.
 - e. Le corps du constructeur doit générer un numéro de compte à l'aide de la méthode **NextNumber** ; définissez la valeur du type de compte à **AccountType.Courant** et initialisez le solde du compte à zéro.

Voici le constructeur terminé :

```
public BankAccount( )
{
    accNo = NextNumber( );
    accType = AccountType.Courant;
    accBal = 0;
}
```

► **Pour créer les autres constructeurs**

1. Ajoutez un autre constructeur prenant un seul paramètre **AccountType** appelé **aType**. Ce constructeur doit :
 - a. Générer un numéro de compte comme le précédent.
 - b. Définir la valeur de *accType* à **aType**.
 - c. Définir la valeur de *accBal* à zéro.
2. Définissez un autre constructeur prenant un seul paramètre **decimal** appelé **aBal**. Ce constructeur doit :
 - a. Générer un numéro de compte.
 - b. Définir la valeur de *accType* à **AccountType.Courant**.
 - c. Définir la valeur de *accBal* à **aBal**.
3. Définissez un dernier constructeur prenant deux paramètres : un **AccountType** appelé **aType** et un **decimal** appelé **aBal**. Ce constructeur doit :
 - a. Générer un numéro de compte.
 - b. Définir la valeur de *accType* à **aType**.
 - c. Définir la valeur de *accBal* à **aBal**.

Voici le code des trois constructeurs :

```
public BankAccount(AccountType aType)
{
    accNo = NextNumber( );
    accType = aType;
    accBal = 0;
}

public BankAccount(decimal aBal)
{
    accNo = NextNumber( );
    accType = AccountType.Courant;
    accBal = aBal;
}

public BankAccount(AccountType aType, decimal aBal)
{
    accNo = NextNumber( );
    accType = aType;
    accBal = aBal;
}
```

► Pour tester les constructeurs

1. Dans la méthode **Main** de la classe **CreateAccount**, définissez quatre variables **BankAccount** appelées *acc1*, *acc2*, *acc3* et *acc4*.
2. Instanciez *acc1* à l'aide du constructeur par défaut.
3. Instanciez *acc2* à l'aide du constructeur prenant seulement un **AccountType**. Définissez la valeur du type de *acc2* à **AccountType.Epargne**.
4. Instanciez *acc3* à l'aide du constructeur prenant seulement un solde **decimal**. Définissez le solde de *acc3* à 100.
5. Instanciez *acc4* à l'aide du constructeur prenant seulement un **AccountType** et un solde **decimal**. Définissez le type de *acc4* à **AccountType.Epargne** et la valeur du solde à 500.
6. Utilisez la méthode **Write** (fournie avec la classe **CreateAccount**) pour afficher le contenu de chaque compte un par un. Le code complet est le suivant :

```
static void Main( )
{
    BankAccount acc1, acc2, acc3, acc4;

    acc1 = new BankAccount( );
    acc2 = new BankAccount(AccountType.Epargne);
    acc3 = new BankAccount(100);
    acc4 = new BankAccount(AccountType.Epargne, 500);

    Write(acc1);
    Write(acc2);
    Write(acc3);
    Write(acc4);
}
```

7. Compilez le projet et corrigez les éventuelles erreurs. Exécutez-le et vérifiez que la sortie est correcte.

Exercice 2

Initialisation de données readonly

Dans cet exercice, vous allez créer une classe appelée **BankTransaction**. Elle contiendra des informations sur une transaction de dépôt ou de retrait effectuée sur un compte.

Dès que le solde d'un compte sera modifié à l'aide de la méthode **Deposit** ou **Withdraw**, un objet **BankTransaction** sera créé. Cet objet contiendra la date et l'heure actuelles (générées à partir de **System.DateTime**) et le montant ajouté (positif) au compte ou retiré (négatif). Les données de transaction ne pouvant pas être modifiées une fois créées, elles seront stockées dans deux variables d'instance **readonly** de l'objet **BankTransaction**.

Le constructeur de **BankTransaction** prendra un seul paramètre décimal, qu'il utilisera pour remplir la variable d'instance du montant de la transaction. La variable d'instance de la date et de l'heure sera remplie par **DateTime.Now**, qui est une propriété de **System.DateTime** retournant la date et l'heure actuelles.

Vous modifierez la classe **BankAccount** pour créer des transactions dans les méthodes **Deposit** et **Withdraw**. Vous stockerez les transactions dans une variable d'instance de la classe **BankAccount** de type **System.Collections.Queue**. Une file d'attente est une structure de données contenant une liste ordonnée d'objets. Elle fournit les méthodes permettant d'ajouter des éléments à la file d'attente et de parcourir la file d'attente. (Il vaut mieux utiliser une file d'attente plutôt qu'un tableau, car sa taille n'est pas fixe : elle s'allonge à mesure que des transactions y sont ajoutées).

► Pour créer la classe **BankTransaction**

1. Ouvrez le projet **Constructors.sln** situé dans le dossier *Fichiers d'atelier\Lab09\Starter\Constructors* si ce n'est déjà fait.
2. Ajoutez-y une nouvelle classe appelée **BankTransaction**.
3. Dans la classe **BankTransaction**, supprimez la directive **namespace** ainsi que la première accolade d'ouverture (**{**) et la dernière accolade de fermeture (**}**). (Vous découvrirez plus en détail les espaces de noms dans un module ultérieur).
4. Dans le commentaire du résumé, ajoutez une brève description de la classe **BankTransaction**. Aidez-vous de la description ci-dessus.
5. Supprimez le constructeur par défaut créé par Microsoft Visual Studio®.
6. Ajoutez les deux variables d'instance privées **readonly** suivantes :
 - a. Une **decimal** appelée *amount*.
 - b. Une variable **DateTime** appelée *when*. La structure **System.DateTime** est utile pour contenir des dates et des heures ; elle contient plusieurs méthodes permettant de manipuler ces valeurs.

7. Ajoutez deux méthodes d'accesseur appelées **Amount** et **When**, qui retournent la valeur des deux variables d'instance :

```
private readonly decimal amount;
private readonly DateTime when;
...
public decimal Amount( )
{
    return amount;
}

public DateTime When( )
{
    return when;
}
```

► Pour créer le constructeur

1. Définissez un constructeur public pour la classe **BankTransaction**. Il prendra un paramètre décimal appelé *tranAmount* qui permettra de remplir la variable d'instance *amount*.
2. Dans le constructeur, initialisez *when* à l'aide de **DateTime.Now**.

Conseil **DateTime.Now** est une propriété, pas une méthode ; vous n'avez par conséquent pas besoin d'utiliser des parenthèses.

Voici le constructeur terminé :

```
public BankTransaction(decimal tranAmount)
{
    amount = tranAmount;
    when = DateTime.Now;
}
```

3. Compilez le projet et corrigez les éventuelles erreurs.

► Pour créer des transactions

1. Comme décrit ci-dessus, la classe **BankAccount** crée des transactions et les stocke dans une file d'attente chaque fois que la méthode **Deposit** ou **Withdraw** est appelée. Retournez à la classe **BankAccount**.
2. Avant le début de la classe **BankAccount**, ajoutez la directive **using** suivante :


```
using System.Collections;
```
3. Ajoutez une variable d'instance privée *tranQueue* à la classe **BankAccount**. Son type de données doit être **Queue** et elle doit être initialisée avec une nouvelle file d'attente vide :


```
private Queue tranQueue = new Queue( );
```

4. Dans la méthode **Deposit**, avant de retourner une valeur, créez une transaction utilisant le montant déposé comme paramètre et ajoutez-la à la file d'attente à l'aide de la méthode **Enqueue**, comme suit :

```
public decimal Deposit(decimal amount)
{
    accBal += amount;
    BankTransaction tran = new BankTransaction(amount);
    tranQueue.Enqueue(tran);
    return accBal;
}
```

5. Dans la méthode **Withdraw**, si les fonds sont suffisants, créez une transaction et ajoutez-la à *tranQueue* comme dans la méthode **Deposit**, comme suit :

```
public bool Withdraw(decimal amount)
{
    bool sufficientFunds = accBal >= amount;
    if (sufficientFunds) {
        accBal += amount;
        BankTransaction tran = new BankTransaction(-amount);
        tranQueue.Enqueue(tran);
    }
    return sufficientFunds;
}
```

Remarque Pour la méthode **Withdraw**, la valeur passée au constructeur de **BankTransaction** doit être constituée du montant du retrait précédé du signe moins.

► Pour tester des transactions

1. Pour les besoins du test, ajoutez une méthode publique appelée **Transactions** à la classe **BankAccount**. Son type de retour doit être **Queue** et la méthode doit retourner *tranQueue*. Vous utiliserez cette méthode pour afficher les transactions à l'étape suivante. La méthode se présentera comme suit :

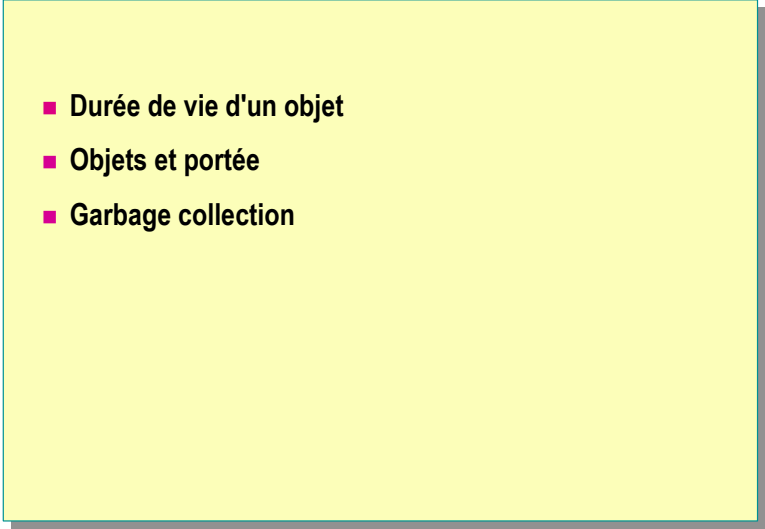
```
public Queue Transactions( )  
{  
    return tranQueue;  
}
```

2. Dans la classe **CreateAccount**, modifiez la méthode **Write** pour afficher les détails des transactions pour chaque compte. Étant donné que les files d'attente implémentent l'interface **IEnumerable**, vous pouvez utiliser la construction **foreach** pour les parcourir par itération.
3. Dans le corps de chaque boucle **foreach**, affichez la date, l'heure et le montant de chaque transaction, en utilisant les méthodes **When** et **Amount** comme suit :

```
static void Write(BankAccount acc)  
{  
    Console.WriteLine("Le numéro du compte est {0}",  
↳acc.Number( ));  
    Console.WriteLine("Le solde du compte est {0}",  
↳acc.Balance( ));  
    Console.WriteLine("Le type du compte est {0}",  
↳acc.Type( ));  
    Console.WriteLine("Transactions :");  
    foreach (BankTransaction tran in acc.Transactions( ))  
    {  
        Console.WriteLine("Date/heure :"); {0}\tMontant : {1}",  
↳tran.When( ), tran.Amount( ));  
    }  
    Console.WriteLine( );  
}
```

4. Dans la méthode **Main**, ajoutez des instructions qui déposeront et retireront de l'argent sur chacun des quatre comptes (*acc1*, *acc2*, *acc3* et *acc4*).
5. Compilez le projet et corrigez les éventuelles erreurs.
6. Exécutez le projet. Examinez la sortie et vérifiez si les transactions s'affichent correctement.

◆ Objets et mémoire

- 
- Durée de vie d'un objet
 - Objets et portée
 - Garbage collection

Dans vos applications, vous aurez besoin de savoir ce qui se passe lorsqu'un objet, plutôt qu'une valeur, devient hors de portée ou est détruit.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- Identifier le rôle du garbage collection lorsqu'un objet devient hors de portée ou est détruit.

Durée de vie d'un objet

■ Création d'objets

- Vous allouez de la mémoire à l'aide de **new**
- Vous initialisez un objet dans cette mémoire à l'aide d'un constructeur

■ Utilisation d'objets

- Vous appelez des méthodes

■ Destruction d'objets

- L'objet est converti en mémoire
- L'allocation de mémoire est annulée

En C#, la destruction d'un objet est un processus à deux étapes qui sont l'inverse des deux étapes nécessaires à la création d'un objet.

Création d'objets

Dans la première section, vous avez appris que la création d'un objet C# pour un type référence se fait en deux étapes :

1. Utilisation du mot clé **new** pour acquérir de la mémoire et l'allouer.
2. Appel d'un constructeur pour transformer la mémoire brute acquise par le mot clé **new** en objet.

Destruction d'objets

La destruction d'un objet C# est également un processus à deux étapes :

1. Désinitialisation de l'objet.

Cette étape convertit l'objet en mémoire brute. On utilise pour ce faire un destructeur C#. Ce processus est l'inverse de l'initialisation qu'effectue le constructeur. Vous pouvez contrôler les opérations qui ont lieu à cette étape en écrivant votre propre destructeur ou méthode finalize.

2. La mémoire brute est désallouée, c'est-à-dire qu'elle est rendue au tas de la mémoire.

Ce processus est l'inverse de l'allocation qu'effectue le mot clé **new**. Vous ne pouvez modifier le comportement de cette étape en aucune manière.

Objets et portée

- **La durée de vie d'une valeur locale est liée à la portée dans laquelle elle est déclarée**
 - Durée de vie courte (en général)
 - Création et destruction déterministes
- **La durée de vie d'un objet dynamique n'est pas liée à sa portée**
 - Durée de vie plus longue
 - Destruction non déterministe

À la différence des valeurs telles que **int** et **struct**, qui sont allouées sur la pile et détruites à la fin de leur portée, les objets sont alloués sur le tas managé et ne sont pas détruits à la fin de leur portée.

Valeurs

La durée de vie d'une valeur locale est liée à la portée dans laquelle elle est déclarée. Les valeurs locales sont des variables allouées sur la pile et non sur le tas managé. Cela signifie que si vous déclarez une variable dont le type est l'un des types primitifs (tel que **int**), **enum** ou **struct**, vous ne pouvez pas l'utiliser en dehors de la portée dans laquelle vous la déclarez. Par exemple, dans l'extrait de code suivant, trois valeurs sont déclarées dans une instruction **for** et sortent de la portée à la fin de cette instruction :

```
struct Point { public int x, y; }
enum Season { Spring, Summer, Fall, Winter }
class Example
{
    void Method(int limit)
    {
        for (int i = 0; i < limit; i++) {
            int x = 42;
            Point p = new Point( );
            Season s = Season.Winter;
            ...
        }
        x = 42;           // Erreur de compilation
        p = new Point( ); // Erreur de compilation
        s = Season.Winter; // Erreur de compilation
    }
}
```

Remarque Dans l'exemple précédent, il semble qu'un **new Point** soit créé. Cependant, puisque **Point** est un **struct**, **new** n'alloue pas de mémoire à partir du tas managé. Le « nouveau » **Point** est créé sur la pile.

Cela signifie que les valeurs locales ont les caractéristiques suivantes :

- Création et destruction déterministes

Une variable locale est créée lorsque vous la déclarez et détruite à la fin de la portée dans laquelle elle est déclarée. Le point de départ et le point de fin de la vie de la valeur sont déterministes, c'est-à-dire qu'ils surviennent à un moment connu.

- Durées de vie généralement très courtes

Vous déclarez une valeur quelque part dans une méthode et cette valeur ne peut exister au-delà de l'appel à la méthode. Lorsque vous retournez une valeur à partir d'une méthode, vous retournez une copie de cette valeur.

Objets

La durée de vie d'un objet n'est pas liée à la portée dans laquelle il est créé. Les objets sont initialisés dans le tas alloué par le biais de l'opérateur **new**. Par exemple, dans le code suivant, la variable de référence *eg* est déclarée dans une instruction **for**. Cela signifie que *eg* sort de la portée à la fin de l'instruction **for** et qu'il s'agit d'une variable locale. Cependant, *eg* est initialisée avec un objet **new Example()** et cet objet ne sort pas de la portée avec *eg*. N'oubliez pas qu'une variable de référence et l'objet qu'elle référence sont deux choses différentes.

```
class Example
{
    void Method(int limit)
    {
        for (int i = 0; i < limit; i++) {
            Example eg = new Example( );
            ...
        }
        // eg est hors de la portée
        // eg existe-t-elle toujours ?
        // L'objet existe-t-il toujours ?
    }
}
```

Cela signifie que les objets ont les caractéristiques suivantes :

- Destruction non déterministe

Un objet est créé lorsque vous le créez mais, à la différence d'une valeur, il n'est pas détruit à la fin de la portée dans laquelle il a été créé. La création d'un objet est déterministe, mais sa destruction ne l'est pas. Vous ne pouvez pas contrôler exactement le moment auquel un objet sera détruit.

- Durée de vie plus longue

La vie d'un objet n'étant pas liée à la méthode qui le crée, l'objet peut exister bien au-delà d'un appel de méthode.

Garbage collection

- **Vous ne pouvez pas détruire explicitement des objets**
 - C# ne fournit pas d'opposé à **new** (**delete**, par exemple)
 - Ceci est dû au fait qu'une fonction de suppression explicite est source de nombreuses erreurs dans d'autres langages
- **Le garbage collection détruit les objets à votre place**
 - Il trouve les objets inaccessibles et les détruit pour vous
 - Il les remet en tas brut inutilisé
 - Il effectue cette tâche lorsque la mémoire devient insuffisante

Jusqu'à présent vous avez vu que vous créez des objets en C# exactement de la même façon que dans d'autres langages, notamment C++. Vous utilisez le mot clé **new** pour allouer de la mémoire à partir du tas et vous appelez un constructeur pour convertir cette mémoire en objet. Cependant, en ce qui concerne la méthode de destruction des objets, il n'existe aucune similitude entre C# et ses prédécesseurs.

Vous ne pouvez pas détruire explicitement des objets

Dans de nombreux langages de programmation, vous pouvez contrôler explicitement le moment où un objet sera détruit. Par exemple, en C++, vous pouvez utiliser une expression **delete** pour désinitialiser (ou finaliser) l'objet (le renvoyer dans la mémoire brute) puis renvoyer la mémoire au tas. En C#, il n'existe aucun moyen de détruire explicitement les objets. À bien des égards, cette restriction est utile, car les programmeurs font souvent un mauvais usage de la capacité à détruire des objets en :

- Oubliant de détruire des objets.

Si vous aviez la responsabilité d'écrire le code détruisant un objet, vous pourriez quelquefois l'oublier. Cela peut se produire dans du code C++ ; il s'agit d'un bogue problématique qui ralentit l'ordinateur de l'utilisateur à mesure que le programme utilise davantage de mémoire. Ce phénomène s'appelle une *fuite de mémoire*. Souvent, le seul moyen de récupérer la mémoire perdue consiste à fermer puis à redémarrer le programme posant des problèmes.

- Tentant de détruire le même objet plusieurs fois.

Il se peut que vous tentiez accidentellement de détruire le même objet plusieurs fois. Cela peut se produire dans du code C++ et il s'agit d'un bogue grave aux conséquences imprévisibles. En effet, lorsque vous détruisez l'objet pour la première fois, la mémoire est récupérée et peut être utilisée pour créer un autre objet, probablement d'une classe complètement différente. Lorsque vous tentez de détruire l'objet pour la seconde fois, la mémoire référence un objet complètement différent.

- Détruisant un objet actif.

Il se peut que vous détruisiez un objet qui est toujours référencé dans une autre partie du programme. Il s'agit également d'un bogue grave, connu sous le nom de *problème de pointeur non résolu*, dont les conséquences sont aussi imprévisibles.

Le garbage collection détruit les objets à votre place

En C#, vous ne pouvez pas détruire explicitement des objets dans le code. C# possède le garbage collection (l'opération de récupération des données par le garbage collector, ou « ramasse-miettes »), qui s'en charge à votre place. Le garbage collection est automatique. Il s'assure des opérations suivantes :

- La destruction des objets.

Il ne précise néanmoins pas exactement le moment où l'objet sera détruit.

- La destruction unique des objets.

Il est ainsi impossible d'aboutir à un comportement imprévisible suite à une double suppression, comme c'est le cas en C++. C'est important, car il est ainsi possible de s'assurer qu'un programme C# se comporte toujours de manière prévisible.

- La destruction des objets inaccessibles uniquement.

Le garbage collection s'assure qu'un objet n'est jamais détruit si un autre objet contient une référence à cet objet. En effet, le garbage collection détruit un objet uniquement si aucun autre objet ne contient une référence à cet objet. La capacité d'un objet à en atteindre un autre par le biais d'une variable de référence s'appelle l'*accessibilité*. Seuls les objets inaccessibles sont détruits. Le rôle du garbage collection consiste à suivre toutes les références d'objets afin de définir quels objets sont accessibles et, en procédant par élimination, de trouver ceux qui ne le sont pas. Cette opération pouvant prendre du temps, le garbage collection ne récupère les données de la mémoire non utilisée que lorsque la quantité de mémoire disponible diminue.

Remarque Vous pouvez forcer le garbage collection explicitement dans votre code, mais cela n'est pas conseillé. Laissez le runtime .NET gérer la mémoire à votre place.

◆ Gestion des ressources

- Nettoyage d'objets
- Écriture d'un destructeur
- Avertissements à propos du minutage du destructeur
- Interface IDisposable et méthode Dispose
- L'instruction using en C#

À la fin de cette section, vous serez à même d'effectuer les tâches suivantes :

- écriture d'un destructeur ;
- utiliser le modèle de conception Dispose.

Nettoyage d'objets

- **Les actions finales des différents objets ne sont pas les mêmes**
 - Elles ne peuvent pas être déterminées par le garbage collection.
 - Les objets du .NET Framework disposent de la méthode **Finalize**.
 - S'il est présent, le garbage collection appellera un destructeur avant de récupérer la mémoire brute.
 - En C#, implémentez un destructeur pour écrire le code de nettoyage. Vous ne pouvez ni appeler, ni substituer **Object.Finalize**.

Vous avez déjà vu que la destruction d'un objet est un processus constitué de deux étapes. Au cours de la première, l'objet est converti en mémoire brute. Au cours de la seconde, la mémoire brute est renvoyée au tas pour y être recyclée. Le garbage collection automatise complètement la seconde étape de ce processus.

Cependant, les actions nécessaires à la finalisation d'un objet spécifique dans la mémoire brute pour le nettoyer dépendent de cet objet. Cela signifie que le garbage collection ne peut automatiser la première étape. Si vous voulez qu'un objet exécute des instructions spécifiques lorsqu'il est sélectionné par le garbage collection juste avant que sa mémoire soit récupérée, vous devez écrire ces instructions vous-même dans un destructeur.

Finalisation

Lorsque le garbage collection détruit un objet inaccessible, il vérifie que la classe de l'objet possède son propre destructeur ou sa méthode **Finalize**. Si tel est le cas, il appelle la méthode avant de recycler la mémoire dans le tas. Les instructions que vous écrivez dans le destructeur ou la méthode **Finalize** sont propres à la classe.

Écriture d'un destructeur

■ Un destructeur est un mécanisme de nettoyage

- Il a sa propre syntaxe :
 - Aucun modificateur d'accès
 - Aucun type de retour, pas même **void**
 - Même nom que celui de la classe, précédé de ~
 - Aucun paramètres

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

Vous pouvez écrire un destructeur pour implémenter le nettoyage d'objets. En C#, la méthode **Finalize** n'est pas directement disponible et vous ne pouvez pas l'appeler ou la remplacer. Vous devez placer le code à exécuter lors de la finalisation dans un destructeur.

L'exemple suivant montre la syntaxe du destructeur C# d'une classe appelée **SourceFile** :

```
~ SourceFile( ) {
// Effectue le nettoyage
}
```

Un destructeur ne possède pas les éléments suivants :

- Modificateur d'accès.
Vous n'appellez pas le destructeur, le garbage collection s'en charge.
- Type de retour.
L'objectif du destructeur n'est pas de retourner une valeur mais d'effectuer les actions de nettoyage requises.
- Paramètres.
Puisque vous n'appellez pas le destructeur, vous ne pouvez pas lui transmettre d'arguments. Notez que cela signifie que le destructeur ne peut pas être surchargé.

Remarque Le compilateur C# convertit automatiquement un destructeur en méthode **Finalize**.

Avertissements à propos du minutage du destructeur

- **L'ordre d'exécution du destructeur et son minutage ne sont pas définis**
 - Il ne s'agit pas nécessairement de l'ordre inverse de la construction
- **Les destructeurs sont toujours appelés**
 - Il ne faut pas se fier au minutage
- **Évitez les destructeurs dans la mesure du possible**
 - Coût en performances
 - Complexité
 - Délai avant la libération de ressources de mémoire

Vous avez vu qu'en C#, le garbage collection se charge de détruire les objets qui sont inaccessibles. Il n'en va pas de même dans d'autres langages, tels que C++, où le programmeur est responsable de la destruction explicite des objets. Il est utile de décharger le programmeur de cette responsabilité, mais en contrepartie, vous ne pouvez pas contrôler le moment où un objet C# est détruit. C'est ce que l'on appelle quelquefois la *finalisation non déterministe*.

L'ordre d'exécution du destructeur et son minutage ne sont pas définis

Dans des langages comme C++, vous pouvez contrôler explicitement le moment où sont créés et détruits les objets. En C#, vous pouvez contrôler l'ordre dans lequel vous créez les objets, mais pas celui dans lequel ils sont détruits. En effet, ce n'est pas vous qui détruisez les objets, mais le garbage collection.

En C#, l'ordre de création des objets ne définit en rien celui de leur destruction. Ils peuvent être détruits dans n'importe quel ordre, et de nombreux autres objets peuvent être détruits entre-temps. Cependant, cela pose rarement problème dans la pratique, car le garbage collection garantit qu'un objet ne sera jamais détruit s'il est accessible. Si un objet contient une référence à un autre objet, le second objet est accessible à partir du premier. Le second objet ne sera donc jamais détruit avant le premier.

Évitez les destructeurs dans la mesure du possible

Évitez autant que possible les destructeurs ou les finaliseurs. En effet, la finalisation augmente la surcharge et la complexité et retarde la récupération des ressources mémoire d'un objet. Implémentez la finalisation ou un destructeur en C# uniquement sur les classes nécessitant une finalisation. Si votre classe ne possède que des ressources managées et qu'elle ne gère pas de ressources autres que la mémoire, vous ne devez pas implémenter de code de finalisation. Le tas pour les objets managés n'est libéré que par le biais du garbage collection.

Interface IDisposable et méthode Dispose

- **Pour récupérer une ressource :**
 - Héritez de l'interface **IDisposable** et implémentez la méthode **Dispose** qui libère les ressources
 - Appelez la méthode **SuppressFinalize**.
 - Assurez-vous de ne pas trop faire appel à **Dispose**
 - N'essayez pas d'utiliser une ressource récupérée

La mémoire est la ressource la plus utilisée par vos programmes et vous pouvez compter sur le garbage collection pour récupérer la mémoire inaccessible lorsque le tas diminue. La mémoire n'est toutefois pas la seule ressource. Votre programme peut notamment utiliser des descripteurs de fichier et des verrous mutex. Ces autres types de ressources sont souvent en quantité plus réduite que la mémoire ou ont besoin d'être libérés rapidement.

Dans ces cas, vous ne pouvez pas compter sur le garbage collection pour les libérer à l'aide d'un destructeur car, comme nous l'avons vu, vous ne pouvez pas savoir quand il va se déclencher. Vous devez plutôt écrire une méthode publique **Dispose** qui libère la ressource puis fait en sorte d'appeler cette méthode au bon moment dans le code. Ces méthodes s'appellent des méthodes Dispose.

En C#, lorsque vous implémentez une méthode **Dispose** :

- Vous héritez de l'interface **IDisposable**.
- Vous implémentez la méthode **Dispose**.
- Vous vous assurez que la méthode **Dispose** puisse être appelée plusieurs fois.
- Vous appelez **SuppressFinalize**.

La méthode **Dispose** d'un type doit libérer toutes les ressources qu'elle possède. Elle doit également libérer toutes les ressources détenues par ses types de base en appelant la méthode **Dispose** de son type parent. La méthode **Dispose** du type parent doit libérer toutes les ressources qu'elle possède et à son tour appeler la méthode **Dispose** de son type parent et transmettre cette procédure dans la hiérarchie des types de base. Pour s'assurer que les ressources sont toujours nettoyées correctement, une méthode **Dispose** doit pouvoir être appelée en toute sécurité plusieurs fois sans lever une exception.

Le code suivant illustre un modèle de conception possible pour l'implémentation d'une méthode **Dispose** pour des classes encapsulant des ressources non managées. Vous trouverez probablement ce modèle facile à utiliser car il est implémenté dans tout le .NET Framework. Il ne s'agit toutefois pas de la seule implémentation possible de la méthode **Dispose**. La classe de base implémente une méthode publique **Dispose** que les utilisateurs de la classe peuvent appeler. Celle-ci appelle à son tour la méthode virtuelle **Dispose** correspondante, selon l'intention de l'appelant. Le code de nettoyage de l'objet est exécuté dans la méthode virtuelle **Dispose**. La classe de base fournit une méthode **Finalize** ou un destructeur comme garantie au cas où la méthode **Dispose** ne serait pas appelée.

```
public class BaseResource: IDisposable
{
    // Pointeur vers une ressource externe
    private IntPtr handle;
    // Autre ressource utilisée par cette classe
    private Component Components;
    // Détermine si Dispose a été appelée
    private bool disposed = false;

    // Constructeur de l'objet BaseResource
    public BaseResource( )
    {
        handle = // Insérez ici du code pour allouer du
        // côté non managé
        Components = new Component (...);
    }

    // Implémente IDisposable.
    // On ne rend pas cette méthode virtuelle.
    // Une classe dérivée ne doit pas être en mesure de
    // remplacer cette méthode.
    public void Dispose( )
    {
        Dispose(true);
        // Se retire de la file d'attente de finalisation
        GC.SuppressFinalize(this);
    }
}
```

Suite du code à la page suivante.

```

protected virtual void Dispose(bool disposing)
{
    // Vérifie si Dispose a déjà été
    // appelée
    if(!this.disposed)
    {
        // S'il s'agit d'un appel à Dispose, élimine toutes
        // les ressources managées
        if(disposing)
        {
            Components.Dispose( );
        }
        // Libère les ressources non managées.
        // Ce n'est pas sécurisé au niveau du thread.
        // Un autre thread peut commencer à éliminer
        // l'objet après la suppression des ressources
        // managées, mais avant que l'indicateur de
        // suppression soit true.
        this.disposed = true;
        Release(handle);
        handle = IntPtr.Zero;
    }
}

// Utilise la syntaxe de destructeur C# pour le code de
// finalisation.
// Ce destructeur s'exécute uniquement si la méthode
// Dispose n'est pas appelée. Cela donne à votre classe de
// base l'occasion de finaliser. Ne
// fournissez pas de destructeurs dans les types dérivés de
// cette classe.
~BaseResource( )
{
    Dispose(false);
}

// Autorise plusieurs appels à votre méthode Dispose,
// mais lève une exception si l'objet a été
// supprimé. Dès que vous réalisez une opération avec
// cette classe, vérifie si elle a été supprimée.
public void DoSomething( )
{
    if(this.disposed)
    {
        throw new ObjectDisposedException(...);
    }
}
}

```

Suite du code à la page suivante.

```
// Modèle de conception d'une classe dérivée.
// Notez que cette classe dérivée implémente par définition
// l'interface IDisposable car elle est implémentée
// dans la classe de base.
public class MyResourceWrapper: BaseResource
{
    private bool disposed = false;

    public MyResourceWrapper( )
    {
        // Constructeur pour cet objet
    }

    protected override void Dispose(bool disposing)
    {
        if(!this.disposed)
        {
            try
            {
                if(disposing)
                {
                    // Libérez ici toutes les ressources managées
                }
                // Libérez ici toutes les ressources non managées
                this.disposed = true;
            }
            finally
            {
                // Appelle Dispose sur votre classe de base.
                base.Dispose(disposing);
            }
        }
    }
}

// Cette classe dérivée ne possède pas de méthode Finalize
// ou de méthode Dispose avec des paramètres car elle les
// hérite de la classe de base.
```

L'instruction using en C#

■ Syntaxe

```
using (Resource r1 = new Resource( ))  
{  
    r1.Method( );  
}
```

■ Dispose est automatiquement appelé à la fin du bloc using

L'instruction **using** en C# définit une portée à la fin de laquelle l'objet sera supprimé.

Vous créez une instance dans une instruction **using** pour vous assurer que **Dispose** est appelée sur l'objet à la fin de l'instruction **using**. Dans l'exemple suivant, **Resource** est un type référence qui implémente **IDisposable** :

```
using (Resource r1 = new Resource( ))  
{  
    r1.Test( );  
}
```

Cette instruction est équivalente dans sa sémantique à la suivante :

```
Resource r1 = new Resource( );  
try {  
    r1.Test( );  
}  
finally {  
    if (r1 != null) ((IDisposable)r1).Dispose( );  
}
```


Atelier 9.2 : Gestion des ressources



Objectifs

Dans cet atelier, vous allez apprendre à utiliser des finaliseurs pour effectuer un traitement avant que le garbage collection ne détruise un objet.

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer un destructeur ;
- créer des requêtes de garbage collection ;
- utiliser le modèle de conception Dispose.

Conditions préalables

Avant de poursuivre, vous devez disposer des connaissances suivantes :

- savoir créer des classes et instancier des objets ;
- savoir définir et appeler des méthodes ;
- définir et utiliser des constructeurs ;
- utiliser la classe **StreamWriter** pour écrire du texte dans un fichier.

Vous devez également avoir réalisé l'atelier 9.1. Dans le cas contraire, vous pouvez utiliser le code de la solution.

Durée approximative de cet atelier : 15 minutes

Exercice 1

Utilisation du modèle de conception Dispose

Dans cet exercice, vous ajouterez une méthode **Dispose** à la classe **BankAccount** pour conserver les données dans le fichier Transaction.dat. La méthode **Dispose** de la classe **BankAccount** parcourt toutes les transactions de la file d'attente et enregistre le journal des transactions dans un fichier.

► Pour créer une méthode Dispose pour la classe BankAccount

1. Ouvrez le projet Finalizers.sln situé dans le dossier *Fichiers d'atelier\Lab09\Starter\Finalizers*.
2. Ajoutez le modificateur **sealed** à la classe **BankAccount** et héritez de l'interface **IDisposable**. Il est impossible d'hériter d'une classe scellée. Le modificateur scellé est ajouté pour simplifier l'implémentation de la méthode Dispose.
3. Ajoutez une variable d'instance **bool** privée appelée *disposed*. Initialisez-la à **false**.
4. Dans la classe **BankAccount**, ajoutez une méthode publique **void** appelée **Dispose** :

```
public void Dispose( )  
{  
  
}
```

5. Dans la méthode **Dispose**, ajoutez des instructions destinées à :
 - a. Examiner la valeur de *disposed*. Si elle est **true**, revenir de la méthode et ne rien faire d'autre.
 - b. Si *disposed* est **false**, créer une variable **StreamWriter** qui ouvre le fichier Transactions.dat dans le répertoire en cours en mode Append (c'est-à-dire qu'elle écrit des données à la fin du fichier s'il existe déjà). Vous pouvez procéder en utilisant la méthode File.AppendText.

```
StreamWriter swFile =  
    File.AppendText("Transactions.Dat");
```
 - c. Utiliser la méthode WriteLine et écrire le numéro de compte, le type et le solde.

```
swFile.WriteLine("Le numéro du compte est {0}", accNo);  
swFile.WriteLine("Le solde du compte est {0}", accBal);  
swFile.WriteLine("Le type du compte est {0}", accType);
```
 - d. Parcourir par itération tous les objets **BankTransaction** dans *tranQueue* et écrire le montant et l'heure de la transaction à l'aide de WriteLine. Faire appel à une instruction **foreach**, comme dans l'atelier 9.1.
 - e. Fermer le **StreamWriter**.
 - f. Définir la valeur de *disposed* à **true**.
 - g. Appeler la méthode **GC.SuppressFinalize**.

Le code complet est le suivant :

```
public void Dispose( )
{
    if (!disposed)
    {
        StreamWriter swFile =
        ↪File.AppendText("Transactions.Dat");
        swFile.WriteLine("Le numéro de compte est {0}",
        ↪accNo);
        swFile.WriteLine("Le solde du compte est {0}",
        ↪accBal);
        swFile.WriteLine("Le type de compte est {0}",
        ↪accType);
        swFile.WriteLine("Transactions :");
        foreach(BankTransaction tran in tranQueue)
        {
            swFile.WriteLine("Date/heure :");
            ↪{0}\tMontant :{1}", tran.When( ), tran.Amount( ));
        }
        swFile.Close( );
        disposed = true;
        GC.SuppressFinalize(this);
    }
}
```

6. Ajouter un destructeur à la classe **BankAccount** qui appelle la méthode **Dispose**.
7. Compiler le projet et corriger d'éventuelles erreurs.

► Pour tester le destructeur

1. Ouvrez le test de validation CreateAccount.cs.
2. Modifiez le code de **Main** pour utiliser l'instruction **using** comme suit :

```
using (BankAccount acc1 = new BankAccount( ))
{
    acc1.Deposit(100);
    acc1.Withdraw(50);
    acc1.Deposit(75);
    acc1.Withdraw(50);
    acc1.Withdraw(30);
    acc1.Deposit(40);
    acc1.Deposit(200);
    acc1.Withdraw(250);
    acc1.Deposit(25);
    Write(acc1);
}
```

3. Compilez le projet et corrigez les éventuelles erreurs.
4. Exécutez le programme.
5. Ouvrez un éditeur de texte et examinez le fichier Transactions.Dat dans le répertoire *Fichiers d'atelier\Lab09\Starter\Finalizers\bin\Debug*.

Contrôle des acquis

- Utilisation de constructeurs
- Initialisation des données
- Objets et mémoire
- Gestion des ressources

-
1. Déclarez une classe que vous appellerez **Date** avec un constructeur public qui attend trois paramètres **int** appelés *year*, *month* et *day*.
 2. Le compilateur va-t-il générer un constructeur par défaut pour la classe **Date** que vous avez déclarée dans la question 1 ? Que se passerait-il si **Date** était un **struct** doté du même constructeur à trois **int** ?

3. Quelle méthode le garbage collection appelle-t-il sur l'objet juste avant de recycler sa mémoire dans le tas ?

4. Quel est le rôle de l'instruction **using** ?

Module 10 : Héritage dans C#

Table des matières

Vue d'ensemble	1
Dérivation de classes	2
Implémentation de méthodes	10
Utilisation de classes scellées (<i>Sealed</i>)	27
Utilisation d'interfaces	29
Utilisation de classes abstraites (<i>Abstract</i>)	42
Atelier 10.1 : Utilisation de l'héritage pour implémenter une interface	52
Contrôle des acquis	71



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Dérivation de classes
- Implémentation de méthodes
- Utilisation de classes scellées (*Sealed*)
- Utilisation d'interfaces
- Utilisation de classes abstraites (*Abstract*)

L'héritage, dans un système orienté objet, est la capacité d'un objet à hériter de données et de fonctionnalités de son objet parent. Par conséquent, un objet enfant peut se substituer à l'objet parent. De plus, en utilisant l'héritage, vous pouvez créer des classes à partir de classes existantes au lieu de tout recommencer et de les créer à partir de rien. Vous pouvez ensuite écrire du code pour ajouter les fonctionnalités requises dans les nouvelles classes. La classe parente sur laquelle la nouvelle classe est basée est appelée une *classe de base* et la classe enfant, la *classe dérivée*.

Lorsque vous créez une classe dérivée, il est important de vous rappeler qu'elle peut se substituer à un type de classe de base. Par conséquent, l'héritage est un mécanisme de classification de types qui s'ajoute au mécanisme de réutilisation du code, le premier étant plus important que le second.

Dans ce module, vous apprendrez à dériver une classe d'une classe de base. Vous allez également apprendre à implémenter des méthodes dans une classe dérivée en les définissant en tant que méthodes virtuelles dans la classe de base et en les substituant ou en les masquant dans la classe dérivée, le cas échéant. Vous apprendrez également à sceller une classe pour qu'il soit impossible d'en dériver une autre. Enfin, vous implémenterez des interfaces et des classes abstraites qui définissent les règles du contrat que les classes dérivées doivent respecter.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- dériver une nouvelle classe d'une classe de base et appeler des membres et des constructeurs de la classe de base à partir de la classe dérivée ;
- déclarer des méthodes comme étant **virtual** et **override** ou les masquer le cas échéant ;
- sceller une classe pour qu'il soit impossible d'en dériver une autre ;
- implémenter des interfaces à l'aide de méthodes implicites et explicites ;
- décrire l'utilisation des classes abstraites et leur implémentation d'interfaces.

◆ Dérivation de classes

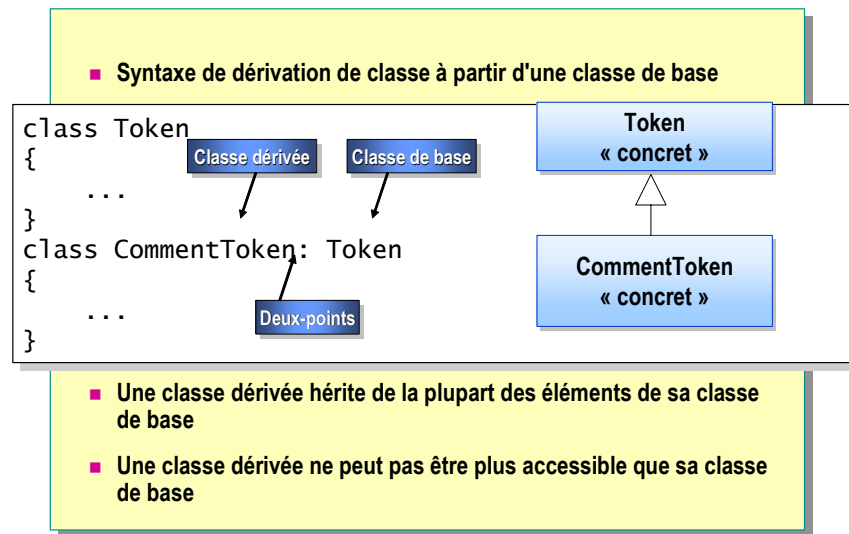
- **Extension des classes de base**
- **Accès aux membres des classes de base**
- **Appel des constructeurs de classe de base**

Vous pouvez dériver une classe uniquement d'une classe de base si celle-ci a été conçue pour autoriser l'héritage. Cela est dû au fait que les objets doivent avoir une structure appropriée pour que l'héritage puisse fonctionner. Une classe de base conçue pour autoriser l'héritage doit le déclarer explicitement. Si une nouvelle classe est dérivée d'une classe de base qui n'est pas conçue de façon appropriée, celle-ci peut changer ultérieurement et rendre la classe dérivée inopérante.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- dériver une nouvelle classe d'une classe de base ;
- accéder aux membres et aux constructeurs de classe de base à partir d'une classe dérivée.

Extension des classes de base



La dérivation d'une classe à partir d'une classe de base est également appelée *l'extension*. Une classe C# peut étendre une seule classe.

Syntaxe de dérivation d'une classe

Pour spécifier qu'une classe est dérivée d'une autre classe, utilisez la syntaxe suivante :

```

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

Les éléments de cette syntaxe sont reproduits sur la diapositive et accompagnés d'une légende. Lorsque vous déclarez une classe dérivée, la classe de base est spécifiée après le signe deux-points. L'espace avant et après le signe deux-points n'est pas important. Il est recommandé, pour cette syntaxe, de n'inclure aucun espace avant le signe deux-points et un seul espace après.

Héritage de classe dérivée

Une classe dérivée hérite de tous les éléments de sa classe de base à l'exception des constructeurs et des destructeurs de classe de base. Les membres publics de la classe de base sont implicitement des membres publics de la classe dérivée. Les membres privés de la classe de base, bien qu'hérités par la classe dérivée, sont accessibles uniquement aux membres de la classe de base.

Accessibilité d'une classe dérivée

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, comme l'illustre le code suivant :

```
class Example
{
    private class NestedBase { }
    public class NestedDerived: NestedBase { } // Erreur
}
```

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.

Accès aux membres des classes de base

```
class Token
{
    ...
    protected string name;
}
class CommentToken: Token{
{
    ...
    public string Name( )
    {
        return name; ✓
    }
}

class Outsidee
{
    void Fails(Token t)
    {
        ...
        t.name ✗
        ...
    }
}
```

- Les membres protégés hérités sont implicitement protégés dans la classe dérivée
- Les méthodes d'une classe dérivée ne peuvent accéder qu'aux membres protégés dont elles ont hérité.
- Les modificateurs d'accès protégés ne peuvent pas être utilisés dans un struct

La signification du modificateur d'accès **protected** dépend de la relation qui existe entre la classe qui contient le modificateur et la classe qui tente d'accéder aux membres qui utilisent ce modificateur.

Les membres d'une classe dérivée peuvent accéder à l'ensemble des membres protégés de leur classe de base. Pour une classe dérivée, le mot clé **protected** se comporte comme le mot clé **public**. C'est pourquoi, dans l'extrait de code présenté sur la diapositive, la méthode **Name** de **CommentToken** peut accéder à la chaîne *name* qui est protégée dans **Token**. Elle est protégée dans **Token**, car **CommentToken** a spécifié **Token** comme sa classe de base.

Cependant, entre les deux classes qui ne sont pas liées par une relation de type classe dérivée à classe de base, les membres protégés d'une classe se comportent comme des membres privés pour l'autre classe. C'est ce pourquoi dans l'extrait de code présenté sur la diapositive, la méthode **Fails** de **Outsidee** ne peut pas accéder à la chaîne *name* protégée dans **Token**, car **Outsidee** ne spécifie pas **Token** comme sa classe de base.

Membres protégés hérités

Lorsqu'une classe dérivée hérite d'un membre protégé, ce membre devient implicitement un membre protégé de la classe dérivée. Cela signifie que les membres protégés sont accessibles à toutes les classes directement ou indirectement dérivées de la classe de base. Ceci est illustré dans l'exemple suivant :

```
class Base
{
    protected string name;
}

class Derived: Base
{
}

class FurtherDerived: Derived
{
    void Compiles( )
    {
        Console.WriteLine(name); // OK
    }
}
```

Membres protégés et méthodes

Les méthodes d'une classe dérivée ne peuvent accéder qu'aux membres protégés dont elles ont hérité. Elles ne peuvent pas accéder aux membres protégés de la classe de base par l'intermédiaire de références à cette classe de base. Ainsi, l'exemple suivant génère une erreur :

```
class CommentToken: Token
{
    void Fails(Token t)
    {
        Console.WriteLine(t.name); // Erreur de compilation
    }
}
```

Conseil Il est souvent recommandé de garder toutes les données comme privées et de n'utiliser les accès protégés qu'avec les méthodes.

Membres protégés et structs

Un **struct** ne prend pas en charge l'héritage. Par conséquent, vous ne pouvez pas dériver d'un **struct** et donc, le modificateur d'accès **protected** ne peut pas être utilisé dans un **struct**. Ainsi, l'exemple suivant génère une erreur :

```
struct Base
{
    protected string name; // Erreur de compilation
}
```

Appel des constructeurs de classe de base

- Les déclarations de constructeurs doivent utiliser le mot clé **base**

```
class Token
{
    protected Token(string name) { .... }
    ...
}
class CommentToken: Token
{
    public CommentToken(string name) : base(name) { }
    ...
}
```

- Il n'est pas possible d'accéder à un constructeur de classe de base privé par une classe dérivée
- Utilisez le mot clé **base** pour qualifier la portée de l'identificateur

Pour appeler un constructeur d'une classe de base à partir du constructeur d'une classe dérivée, utilisez le mot clé **base**. La syntaxe pour ce mot clé est la suivante :

```
C(...): base( ) {...}
```

Le signe deux-points et l'appel du constructeur de classe de base associé sont tous deux appelés des *initialiseurs de constructeur*.

Déclarations de constructeurs

Si la classe dérivée n'appelle pas explicitement un constructeur de classe de base, le compilateur C# utilise implicitement un initialiseur de constructeur : **base()**. Cela implique qu'une déclaration de constructeur de ce type :

```
C(...) {...}
```

est équivalent à

```
C(...): base( ) {...}
```

Ce comportement implicite est souvent parfaitement adéquat pour les raisons suivantes :

- Une classe sans classe de base explicite étend implicitement la classe **System.Object** qui contient un constructeur public sans paramètre.
- Si une classe ne contient pas de constructeur, le compilateur fournit automatiquement un constructeur public sans paramètre appelé le constructeur par défaut.

Si une classe fournit son propre constructeur explicite, le compilateur ne crée pas de constructeur par défaut. Cependant, si le constructeur spécifié ne correspond à aucun constructeur de la classe de base, le compilateur génère une erreur, comme l'illustre le code suivant :

```
class Token
{
    protected Token(string name) { ... }
}

class CommentToken: Token
{
    public CommentToken(string name) { ... } // Erreur
}
```

L'erreur se produit parce que le constructeur **CommentToken** contient implicitement un initialiseur de constructeur : `base()` et que la classe de base **Token** ne contient pas de constructeur sans paramètre. Vous pouvez corriger cette erreur en utilisant le code de la diapositive.

Règles d'accès d'un constructeur

Les règles d'accès d'un constructeur dérivé pour appeler un constructeur de classe de base sont exactement les mêmes que celles des méthodes régulières. Par exemple, si le constructeur de classe de base est privé (private) alors la classe dérivée ne peut pas y accéder :

```
class NonDerivable
{
    private NonDerivable( ) { ... }
}

class Impossible: NonDerivable
{
    public Impossible( ) { ... } // Erreur de compilation
}
```

Dans ce cas, une classe dérivée ne peut pas appeler le constructeur de classe de base.

Portée d'un identificateur

Vous pouvez également utiliser le mot clé **base** pour qualifier la portée d'un identificateur. Cela peut s'avérer utile étant donné qu'une classe dérivée est autorisée à déclarer des membres qui ont le même nom que des membres de classes de base. Le code suivant fournit un exemple :

```
class Token
{
    protected string name;
}
class CommentToken: Token
{
    public void Method(string name)
    {
        base.name = name;
    }
}
```

Remarque Contrairement à ce qui se fait en C++, le nom de la classe de base, tel que **Token** dans l'exemple de la diapositive, n'est pas utilisé. Le mot clé **base** fait référence sans ambiguïté à la classe de base, car en C# une classe ne peut étendre qu'une seule classe de base.

◆ Implémentation de méthodes

- Définition de méthodes virtuelles
- Utilisation des méthodes virtuelles
- Substitution de méthodes
- Utilisation des méthodes override
- Utilisation de new pour masquer des méthodes
- Utilisation du mot clé new
- Application pratique : Implémentation de méthodes
- Quiz : Trouver les bogues

Vous pouvez redéfinir les méthodes d'une classe de base dans une classe dérivée lorsque les méthodes de la classe de base ont été conçues pour la substitution.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- utiliser le type de méthode **virtual** ;
- utiliser le type de méthode **override** ;
- utiliser le type de méthode **hide**.

Définition de méthodes virtuelles

■ Syntaxe : Déclarez-les comme étant virtuelles (virtual)

```
class Token
{
    ...
    public int LineNumber( )
    { ...
    }
    public virtual string Name( )
    { ...
    }
}
```

■ Les méthodes virtuelles sont polymorphes

Une méthode virtuelle spécifie *une* implémentation possible d'une méthode qui peut être substituée par polymorphisme dans une classe dérivée. En revanche, une méthode non virtuelle spécifie *l'unique* implémentation possible d'une méthode. Il est impossible de substituer par polymorphisme une méthode non virtuelle dans une classe dérivée.

Remarque En C#, la présence ou l'absence d'une méthode virtuelle permet de savoir si l'auteur de cette méthode l'a conçue ou non pour être utilisée comme classe de base.

Syntaxe du mot clé

Pour déclarer une méthode virtuelle, vous devez utiliser le mot clé **virtual**. La syntaxe de ce mot clé est présentée sur la diapositive.

Lorsque vous déclarez une méthode virtuelle, elle doit contenir un corps de méthode. Si elle ne contient pas de corps, le compilateur génère une erreur, comme illustré ci-dessous :

```
class Token
{
    public virtual string Name( ); // Erreur de compilation
}
```

Utilisation des méthodes virtuelles

■ Pour utiliser des méthodes virtuelles :

- Vous ne pouvez pas déclarer des méthodes virtuelles comme étant statiques
- Vous ne pouvez pas déclarer des méthodes virtuelles comme étant privées

Pour utiliser des méthodes virtuelles de façon efficace, vous devez comprendre les points suivants :

- Vous ne pouvez pas déclarer des méthodes virtuelles comme étant statiques ;
vous ne pouvez pas qualifier des méthodes virtuelles comme étant statiques, car les méthodes statiques sont des méthodes de classe et que le polymorphisme fonctionne avec des objets et non des classes ;
- Vous ne pouvez pas déclarer des méthodes virtuelles comme étant privées ;
Il est impossible de déclarer des méthodes virtuelles comme étant privées, car elles ne peuvent pas être substituées par polymorphisme dans une classe dérivée. Le code suivant fournit un exemple :

```
class Token
{
    private virtual string Name( ) { ... }
    // Erreur de compilation
}
```

Substitution de méthodes

■ Syntaxe : Utilisez le mot clé **override**

```
class Token
{
    ...
    public virtual string Name( ) { .... }
}
class CommentToken: Token
{
    ...
    public override string Name( ) { .... }
}
```

Une méthode **override** (de substitution) spécifie *une autre* implémentation d'une méthode virtuelle. Les méthodes virtuelles sont définies dans une classe de base et peuvent être substituées par polymorphisme dans une classe dérivée.

Syntaxe du mot clé

Pour déclarer une méthode **override**, vous devez utiliser le mot clé **override**, comme l'illustre le code suivant :

```
class Token
{
    ...
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{
    ...
    public override string Name( ) { ... }
}
```

Comme pour les méthodes virtuelles, vous devez inclure un corps de méthode dans une méthode **override**, afin que le compilateur ne génère pas d'erreurs. Le code suivant fournit un exemple :

```
class Token
{
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{
    public override string Name( ); // Erreur de compilation
}
```

Utilisation des méthodes override

- Seules peuvent être substituées les méthodes virtuelles héritées identiques

```
class Token
{
    ...
    public int LineNumber( ) { .... }
    public virtual string Name( ) { .... }
}
class CommentToken: Token
{
    ...
    public override int LineNumber( ) { .... } ✗
    public override string Name( ) { .... } ✓
}
```

- Vous devez faire correspondre une méthode override à sa méthode virtuelle associée
- Vous pouvez substituer une méthode override
- Vous ne pouvez pas déclarer explicitement une méthode override comme virtuelle
- Vous ne pouvez pas déclarer une méthode override comme étant statique ou privée

Pour utiliser les méthodes override de façon efficace, vous devez comprendre les quelques restrictions suivantes :

- Seules peuvent être substituées les méthodes virtuelles héritées identiques.
- Vous devez faire correspondre une méthode override à sa méthode virtuelle associée.
- Vous pouvez substituer une méthode override.
- Vous ne pouvez pas déclarer explicitement une méthode override comme virtuelle.
- Vous ne pouvez pas déclarer une méthode override comme étant statique ou privée.

Chacune de ces restrictions est décrite plus en détail dans les sections qui suivent.

Seules peuvent être substituées les méthodes virtuelles héritées identiques

Vous pouvez utiliser une méthode override uniquement pour substituer une méthode virtuelle héritée identique. Dans le code de la diapositive, la méthode **LineNumber** de la classe dérivée **CommentToken** entraîne une erreur de compilation, car la méthode héritée **Token.LineNumber** n'est pas marquée comme étant virtuelle.

Vous devez faire correspondre une méthode override à sa méthode virtuelle associée

Une déclaration override doit être identique en tous points à la méthode virtuelle à laquelle elle se substitue. Elles doivent toutes deux avoir le même niveau d'accès, le même type de retour, le même nom et les mêmes paramètres.

Par exemple, la substitution échoue dans l'exemple suivant, car les niveaux d'accès sont différents (**protected** et **public**), les types de retour sont différents (**string** et **void**) et les paramètres sont également différents (**none** et **int**) :

```
class Token
{
    protected virtual string Name( ) { ... }
}
class CommentToken: Token
{
    public override void Name(int i) { ... } // Erreurs
}
```

Vous pouvez substituer une méthode override

Une méthode override est implicitement virtuelle et vous pouvez donc la substituer. Le code suivant fournit un exemple :

```
class Token
{
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{
    public override string Name( ) { ... }
}
class OneLineCommentToken: CommentToken
{
    public override string Name( ) { ... } // OK
}
```

Vous ne pouvez pas déclarer explicitement une méthode override comme virtuelle

Une méthode override est implicitement virtuelle, mais ne peut pas être explicitement qualifiée de virtuelle. Le code suivant fournit un exemple :

```
class Token
{
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{
    public virtual override string Name( ) { ... } // Erreur
}
```

Vous ne pouvez pas déclarer une méthode override comme étant statique ou privée

Une méthode override ne peut jamais être qualifiée comme étant statique, car les méthodes statiques sont des méthodes de classe et que le polymorphisme fonctionne avec des objets et non avec des classes.

Par ailleurs, une méthode override ne peut jamais être privée. La raison est due au fait qu'une méthode override doit se substituer à une méthode virtuelle et qu'une méthode virtuelle ne peut pas être privée.

Utilisation de new pour masquer des méthodes

- **Syntaxe : Utilisez le mot clé new pour masquer une méthode**

```
class Token
{
    ...
    public int LineNumber( ) { .... }
}
class CommentToken: Token
{
    ...
    new public int LineNumber( ) { .... }
}
```

Vous pouvez masquer une méthode héritée identique en ajoutant une nouvelle méthode dans la hiérarchie de classes. L'ancienne méthode dont la classe dérivée a hérité à partir de la classe de base est ensuite remplacée par une méthode complètement différente.

Syntaxe du mot clé

Utilisez le mot clé **new** pour masquer une méthode. La syntaxe pour ce mot clé est la suivante :

```
class Token
{
    ...
    public int LineNumber( ) { ... }
}
class CommentToken: Token
{
    ...
    new public int LineNumber( ) { ... }
}
```

Utilisation du mot clé new

■ Masquer les méthodes virtuelles et non virtuelles

```
class Token
{
    ...
    public int LineNumber( ) { .... }
    public virtual string Name( ) { .... }
}
class CommentToken: Token
{
    ...
    new public int LineNumber( ) { .... }
    public override string Name( ) { .... }
}
```

■ Résoudre les conflits de noms dans le code

■ Masquer des méthodes qui ont des signatures identiques

L'utilisation du mot clé **new** permet d'effectuer les opérations suivantes :

- Masquer des méthodes virtuelles et non virtuelles.
- Résoudre les conflits de noms dans le code.
- Masquer des méthodes qui ont des signatures identiques.

Chacune de ces opérations est décrite en détail dans les sous-sections qui suivent.

Masquer des méthodes virtuelles et non virtuelles

L'utilisation du mot clé **new** pour masquer une méthode a des implications si vous utilisez le polymorphisme. Par exemple, dans le code de la diapositive, **CommentToken.LineNumber** est une méthode **new**. Elle n'a absolument aucun lien avec la méthode **Token.LineNumber**. Même si la méthode **Token.LineNumber** était une méthode virtuelle, la méthode **CommentToken.LineNumber** serait toujours une méthode **new** sans aucun rapport.

Dans cet exemple, la méthode **CommentToken.LineNumber** n'est pas virtuelle. Cela signifie qu'une classe dérivée ne peut pas se substituer à la méthode **CommentToken.LineNumber**. Toutefois, la méthode **new CommentToken.LineNumber** peut être déclarée comme virtuelle et dans ce cas, des classes dérivées peuvent s'y substituer.

```
class CommentToken: Token
{
    ...
    new public virtual int LineNumber( ) { ... }
}
class OneLineCommentToken: CommentToken
{
    public override int LineNumber( ) { ... }
}
```

Conseil Voici la syntaxe recommandée pour les méthodes virtuelles new :

```
new public virtual int LineNumber( ) { ... }
de préférence à :
public new virtual int LineNumber( ) { ... }
```

Résoudre les conflits de noms dans le code

Les conflits de noms génèrent souvent des avertissements pendant la compilation. Considérons par exemple le code suivant :

```
class Token
{
    public virtual int LineNumber( ) { ... }
}
class CommentToken: Token
{
    public int LineNumber( ) { ... }
}
```

Si vous compilez ce code, vous recevrez un message d'avertissement qui signale que **CommentToken.LineNumber** masque **Token.LineNumber**. Ce message met en évidence le nom à l'origine du conflit. Vous pouvez alors choisir parmi trois options :

1. Ajouter un qualificateur **override** à **CommentToken.LineNumber**.
2. Ajouter un qualificateur **new** à la méthode **CommentToken.LineNumber**. Dans ce cas, la méthode cache toujours la méthode identique dans la classe de base, mais le qualificateur **new** indique explicitement au compilateur et aux personnes chargées de la gestion du code que le conflit de nom n'est pas accidentel.
3. Modifier le nom de la méthode.

Masquer des méthodes qui ont des signatures identiques

Le modificateur **new** est nécessaire uniquement lorsqu'une méthode de classe dérivée masque une méthode de classe de base visible qui a une signature identique. Dans l'exemple qui suit, le compilateur signale que le modificateur **new** n'est pas nécessaire, car les méthodes prennent des paramètres différents et n'ont donc pas de signatures identiques :

```
class Token
{
    public int LineNumber(short s) { ... }
}
class CommentToken: Token
{
    new public int LineNumber(int i) { ... } // Avertissement
}
```

À l'inverse, si les deux méthodes ont des signatures identiques, le compilateur signale que le modificateur **new** est nécessaire, car la méthode de la classe de base est masquée. Dans l'exemple qui suit, les deux méthodes ont des signatures identiques parce que les types de retour ne font pas partie de la signature d'une méthode :

```
class Token
{
    public virtual int LineNumber( ) { ... }
}
class CommentToken: Token
{
    public void LineNumber( ) { ... } // Avertissement
}
```

Remarque Vous pouvez aussi utiliser le mot clé **new** pour masquer des champs et des classes imbriquées.

Application pratique : Implémentation de méthodes

```
class A {  
    public virtual void M() { Console.Write("A"); }  
}  
class B: A {  
    public override void M() { Console.Write("B"); }  
}  
class C: B {  
    new public virtual void M() { Console.Write("C"); }  
}  
class D: C {  
    public override void M() { Console.Write("D"); }  
    static void Main( ){  
        D d = new D(); C c = d; B b = c; A a = b;  
        d.M(); c.M(); b.M(); a.M();  
    }  
}
```

Pour vous exercer à utiliser les mots clés **virtual**, **override** et **new**, parcourez le code présenté sur cette diapositive et essayez d'imaginer ce qu'il permet d'obtenir une fois compilé.

Solution

Après l'exécution du programme, le résultat affiche DDBB sur la console.

Logique du programme

Un seul objet est créé par le programme. Il s'agit de l'objet de type **D** créé dans la déclaration suivante :

```
D d = new D( );
```

Les instructions de déclaration restantes dans **Main** déclarent des variables de différents types qui font tous référence à ce même objet :

- c est une référence **C** à d.
- b est une référence **B** à c, qui est une référence à d.
- a est une référence **A** à b, qui est une référence à c, lui-même référence à d.

Suivent enfin les quatre instructions d'expression. Elles sont expliquées individuellement ci-après.

Première instruction :

```
d.M( )
```

Il s'agit de l'appel à **D.M**, qui est déclarée comme étant une méthode override et qui est par conséquent implicitement virtuelle. Cela signifie qu'au moment de l'exécution, le compilateur appelle l'implémentation la plus dérivée de **D.M** dans l'objet du type **D**. Cette implémentation est **D.M** et D s'affiche sur la console.

Deuxième instruction :

`c.M()`

Il s'agit de l'appel à **C.M**, qui est déclarée comme étant une méthode virtuelle. Cela signifie qu'au moment de l'exécution, le compilateur appelle l'implémentation la plus dérivée de **C.M** dans l'objet du type **D**. Étant donné que **D.M** se substitue à **C.M**, **D.M** est l'implémentation la plus dérivée. Par conséquent **D.M** est appelée et D s'affiche à nouveau sur la console.

Troisième instruction :

`b.M()`

Il s'agit de l'appel à **B.M**, qui est déclarée comme étant une méthode override et qui est par conséquent implicitement virtuelle. Cela signifie qu'au moment de l'exécution, le compilateur appelle l'implémentation la plus dérivée de **B.M** dans l'objet du type **D**. Étant donné que **C.M** ne se substitue pas à **B.M**, mais qu'elle introduit une méthode **new** qui *masque* **C.M**, l'implémentation la plus dérivée de **B.M** dans l'objet du type **D** est **B.M**. Par conséquent **B.M** est appelée et B s'affiche sur la console.

Quatrième instruction :

`a.M()`

Il s'agit de l'appel à **A.M**, qui est déclarée comme étant une méthode virtuelle. Cela signifie qu'au moment de l'exécution, le compilateur appelle l'implémentation la plus dérivée de **A.M** dans l'objet du type **D**. **B.M** se substitue à **A.M**, mais comme précédemment, **C.M** ne se substitue pas à **B.M**. Par conséquent, l'implémentation la plus dérivée de **A.M** dans l'objet du type **D** est **B.M**. **B.M** est donc appelée et B s'affiche à nouveau sur la console.

C'est ainsi que le programme génère le résultat DDBB et l'affiche sur la console.

Dans cet exemple, les classes **C** et **D** contiennent deux méthodes **virtual** qui ont la même signature : celle introduite par **A** et celle introduite par **C**. La méthode introduite par **C** masque celle introduite par **A**. De ce fait, la déclaration **override** dans **D** se substitue à la méthode introduite par **C**, et **D** ne peut pas se substituer à la méthode introduite par **A**.

Quiz : Trouver les bogues

```
class Base
{
    public void Alpha( ) { .... }
    public virtual void Beta( ) { .... }
    public virtual void Gamma(int i) { .... }
    public virtual void Delta( ) { .... }
    private virtual void Epsilon( ) { .... }
}
class Derived: Base
{
    public override void Alpha( ) { .... }
    protected override void Beta( ) { .... }
    public override void Gamma(double d) { .... }
    public override int Delta( ) { .... }
}
```

Vous pouvez travailler avec un partenaire pour trouver les bogues du code reproduit sur la diapositive. Les réponses se trouvent à la page suivante.

Réponses

Les erreurs suivantes se produisent dans ce code :

1. La classe **Base** déclare une méthode privée virtuelle appelée **Epsilon**. Les méthodes privées ne peuvent pas être virtuelles. Le compilateur C# intercepte ce bogue comme erreur de compilation. Vous pouvez corriger le code comme suit :

```
class Base
{
    ...
    public virtual void Epsilon( ) { ... }
}
```

ou bien, le corriger ainsi :

```
class Base
{
    ...
    private void Epsilon( ) { ... } // Non virtuelle
}
```

2. La classe **Derived** déclare la méthode **Alpha** avec le modificateur **override**. Cependant, la méthode **Alpha** dans la classe de base n'est pas marquée comme étant virtuelle. Vous ne pouvez substituer qu'une méthode virtuelle. Le compilateur C# intercepte ce bogue comme erreur de compilation. Vous pouvez corriger le code comme suit :

```
class Base
{
    public virtual void Alpha( ) { ... }
    ...
}
```

ou bien, le corriger ainsi :

```
class Derived: Base
{
    /* tout */ new void Alpha( ) { ... }
    ...
}
```


3. La classe **Derived** déclare une méthode protégée appelée **Beta** avec le modificateur **override**. Cependant, la méthode de la classe de base **Beta** est publique. Lorsque vous substituez une méthode, vous ne pouvez pas modifier son accès. Le compilateur C# intercepte ce bogue comme erreur de compilation. Vous pouvez corriger le code comme suit :

```
class Derived: Base
{
    ...
    public override void Beta( ) { ... }
    ...
}
```

ou bien, le corriger ainsi :

```
class Derived: Base
{
    ...
    /* tout accès */ new void Beta( ) { ... }
    ...
}
```

4. La classe **Derived** déclare une méthode publique appelée **Gamma** avec le modificateur **override**. Cependant, la méthode de la classe de base appelée **Gamma** et la méthode de la classe **Derived** appelée **Gamma** prennent des paramètres différents. Lorsque vous substituez une méthode, vous ne pouvez pas modifier les types de paramètres. Le compilateur C# intercepte ce bogue comme erreur de compilation. Vous pouvez corriger le code comme suit :

```
class Derived: Base
{
    ...
    public override void Gamma(int i) { ... }
}
```

ou bien, le corriger ainsi :

```
class Derived: Base
{
    ...
    /* tout accès */ void Gamma(double d) { ... }
    ...
}
```

5. La classe **Derived** déclare une méthode publique appelée **Delta** avec le modificateur **override**. Cependant, la méthode de la classe de base appelée **Delta** et la méthode de la classe dérivée appelée **Delta** retournent des types différents. Lorsque vous substituez une méthode, vous ne pouvez pas modifier le type de retour. Le compilateur C# intercepte ce bogue comme erreur de compilation. Vous pouvez corriger le code comme suit :

```
class Derived: Base
{
    ...
    public override void Delta( ) { ... }
}
```

ou bien, le corriger ainsi :

```
class Derived: Base
{
    ...
    /* tout accès */ new int Delta( ) { ... }
    ...
}
```

Utilisation de classes scellées (*Sealed*)

- Il est impossible de dériver une classe d'une classe scellée
- Vous pouvez utiliser des classes scellées pour optimiser les opérations au moment de l'exécution
- Beaucoup de classes du .NET Framework sont scellées : `String`, `StringBuilder`, etc.
- Syntaxe : Utilisez le mot clé `sealed`

```
namespace System
{
    public sealed class String
    {
        ...
    }
}
namespace Mine
{
    class FancyString: String { .... } ❌
}
```

La création d'une hiérarchie d'héritage souple n'est pas une tâche aisée.

La plupart des classes sont autonomes et ne sont pas conçues pour autoriser la dérivation d'autres classes. Cependant, en termes de syntaxe, la dérivation à partir d'une classe est très simple et s'effectue au moyen de quelques touches du clavier. Cela représente un danger pour les programmeurs qui sont tentés de dériver des classes à partir de classes qui ne sont pas conçues pour jouer le rôle de classe de base.

Pour contourner ce problème et permettre aux programmeurs de mieux exprimer leurs intentions au compilateur et aux autres programmeurs, C# autorise la déclaration de classes *scellées* (*sealed*). Il est impossible de dériver une classe d'une classe scellée.

Syntaxe du mot clé

Pour sceller une classe, utilisez le mot clé **sealed**. La syntaxe pour ce mot clé est la suivante :

```
namespace System
{
    public sealed class String
    {
        ...
    }
}
```

L'environnement Microsoft® .NET Framework contient de nombreux exemples de classes scellées. La diapositive représente la classe **System.String**, où le mot clé **string** est l'alias de cette classe. Cette classe est scellée et n'autorise donc pas la dérivation.

Optimisation des opérations à l'exécution

Le modificateur **sealed** permet certaines optimisations au moment de l'exécution. Une classe scellée ne pouvant pas avoir de classes dérivées, il est possible de transformer des appels de fonctions membres virtuelles sur des instances de classes scellées en appels de fonctions membres non virtuelles.

◆ Utilisation d'interfaces

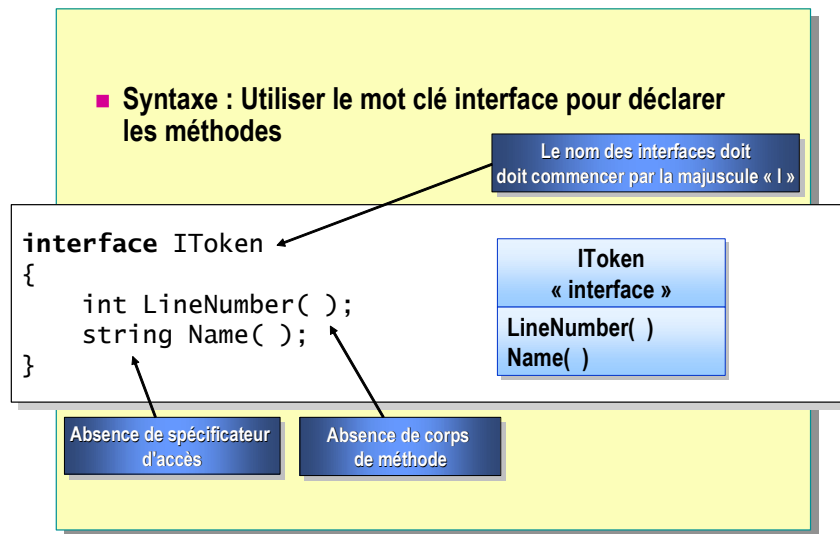
- Déclaration d'interfaces
- Implémentation d'interfaces multiples
- Implémentation de méthodes d'interface
- Implémentation explicite de méthodes d'interface
- Quiz : Trouver les bogues

Une interface spécifie un contrat syntaxique et sémantique auquel toutes les classes dérivées doivent adhérer. Plus précisément, une interface décrit la partie *objet* du contrat et les classes qui implémentent l'interface décrivent la partie *moyen* du contrat.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- utiliser la syntaxe pour déclarer des interfaces ;
- utiliser les deux techniques pour implémenter des méthodes d'interface dans des classes dérivées.

Déclaration d'interfaces



Une interface ressemble à une classe sans code. La déclaration d'une interface s'effectue de la même façon que la déclaration d'une classe. Pour déclarer une interface en C#, utilisez le mot clé **interface** au lieu de **class**. La syntaxe de ce mot clé est expliquée sur la diapositive.

Remarque Il est recommandé de faire précéder tous les noms d'interface de la lettre majuscule « I ». Par exemple, utilisez **IToken** plutôt que **Token**.

Caractéristiques des interfaces

Parmi les caractéristiques des interfaces, les deux suivantes sont les plus importantes.

Les méthodes d'interface sont implicitement publiques

Les méthodes déclarées dans une interface sont implicitement publiques. Par conséquent, les modificateurs d'accès **public** explicites ne sont pas autorisés, comme le montre l'exemple suivant :

```
interface IToken
{
    public int LineNumber( ); // Erreur de compilation
}
```

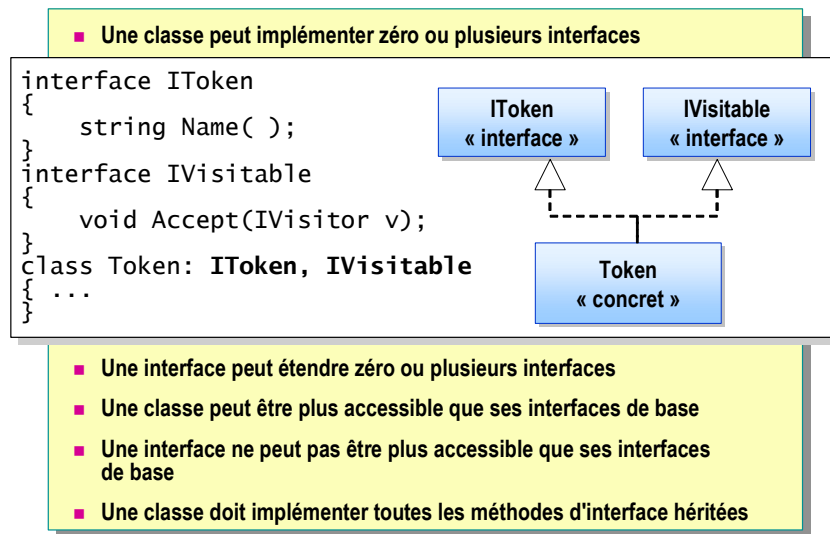
Les méthodes d'interface ne contiennent pas de corps de méthode

Les méthodes déclarées dans une interface ne sont pas autorisées à contenir des corps de méthode. Par exemple, le code suivant n'est pas autorisé :

```
interface IToken
{
    int LineNumber( ) { ... } // Erreur de compilation
}
```

À proprement parler, les interfaces peuvent contenir des déclarations de propriété d'interface, qui sont des déclarations de propriétés sans corps, des déclarations d'événement d'interface, qui sont des déclarations d'événements sans corps et des déclarations d'indexeur d'interface, qui sont des déclarations d'indexeurs sans corps.

Implémentation d'interfaces multiples



Bien que C# autorise seulement l'héritage simple, il permet d'implémenter plusieurs interfaces dans une même classe. Cette section traite des différences entre une classe et une interface en termes respectivement d'implémentation et d'extension d'interfaces, et aborde par ailleurs, leur accessibilité comparativement à leurs interfaces de base.

Implémentation des interfaces

Une classe peut implémenter zéro ou plusieurs interfaces mais elle ne peut étendre explicitement qu'une seule classe. Cet aspect est illustré sur la diapositive.

Remarque À proprement parler, une classe étend toujours une classe. Si vous ne spécifiez pas de classe de base, votre classe héritera implicitement de la classe **object**.

Par opposition, une interface peut étendre zéro ou plusieurs interfaces. Par exemple, vous pouvez réécrire le code de la diapositive de la façon suivante :

```
interface IToken { ... }
interface IVisitable { ... }
interface IVisitableToken: IVisitable, IToken { ... }
class Token: IVisitableToken { ... }
```

Accessibilité

Une classe peut être plus accessible que ses interfaces de base. Par exemple, vous pouvez déclarer une classe publique qui implémente une interface privée comme suit :

```
class Example
{
    private interface INested { }
    public class Nested: INested { } // OK
}
```

Cependant, une interface ne peut pas être plus accessible que ses interfaces de base. C'est une erreur de déclarer une interface publique qui étend une interface privée, comme le montre l'exemple qui suit :

```
class Example
{
    private interface INested { }

    public interface IAlsoNested: INested { }
    // Erreur de compilation
}
```

Méthodes d'interface

Une classe doit implémenter toutes les méthodes de toutes les interfaces qu'elle étend, que ces interfaces soient héritées directement ou indirectement.

Implémentation de méthodes d'interface

- La méthode d'implémentation doit être la même que la méthode d'interface
- La méthode d'implémentation peut être virtuelle ou non virtuelle

```
class Token: IToken, IVisitable
{
    public virtual string Name( )
    { ...
    }
    public void Accept(IVisitor v)
    { ...
    }
}
```

Même accès
Même type de retour
Même nom
Mêmes paramètres

Lorsqu'une classe implémente une interface, elle doit implémenter toutes les méthodes déclarées dans cette interface. Cette règle est pratique parce que les interfaces ne peuvent pas définir leurs propres corps de méthode.

La méthode que la classe implémente doit être identique en tout point à la méthode d'interface. Elle doit avoir les mêmes :

- Accès

Étant donné qu'une méthode d'interface est implicitement publique, cela signifie que la méthode d'implémentation doit être explicitement déclarée comme étant publique. Si le modificateur d'accès est omis, alors la méthode est privée par défaut.

- Type de retour

Si le type de retour dans l'interface est déclaré comme étant du type **T**, alors le type de retour dans la classe d'implémentation ne peut pas être déclaré comme étant un type dérivé de **T** ; il doit être du type **T**. En d'autres termes, la covariance des types de retour n'est pas acceptée en C#.

- Nom

En C#, les noms respectent la casse.

- Liste de types de paramètres

Le code suivant respecte l'ensemble de ces règles :

```
interface IToken
{
    string Name( );
}
interface IVisitable
{
    void Accept(IVisitor v);
}
class Token: IToken, IVisitable
{
    public virtual string Name( )
    { ...
    }
    public void Accept(IVisitor v)
    { ...
    }
}
```

La méthode d'implémentation peut être virtuelle, comme la méthode **Name** dans le code précédent. Dans ce cas, la méthode peut être substituée dans des classes à dériver. La méthode d'implémentation peut également être non virtuelle, comme la méthode **Accept** dans le code précédent. Dans ce cas, la méthode ne peut pas être substituée dans des classes à dériver.

Implémentation explicite de méthodes d'interface

■ Utilisation du nom qualifié complet de la méthode d'interface

```
class Token: IToken, IVisitable
{
    string IToken.Name( )
    { ...
    }
    void IVisitable.Accept(IVisitor v)
    { ...
    }
}
```

■ Restrictions relatives à l'implémentation explicite des méthodes d'interface

- Il n'est possible d'accéder aux méthodes que par l'interface
- Il est impossible de déclarer des méthodes comme étant virtuelles
- Il est impossible de spécifier un modificateur d'accès

Cette technique qui permet à une classe d'implémenter une méthode héritée d'une interface consiste à utiliser l'implémentation explicite des méthodes d'interface.

Utilisation du nom qualifié complet de la méthode d'interface

Lorsque vous implémentez explicitement une méthode d'interface, vous devez utiliser le nom qualifié complet de la méthode d'implémentation. Cela implique que le nom de la méthode comporte le nom de l'interface, comme **IToken** dans **IToken.Name**.

La diapositive montre un exemple d'implémentation explicite de deux méthodes d'interface par la classe **Token**. Notez les différences entre cette implémentation et l'implémentation précédente.

Restrictions relatives à l'implémentation explicite des méthodes d'interface

Lorsque vous implémentez des interfaces explicites, vous devez tenir compte des restrictions suivantes.

Il n'est possible d'accéder aux méthodes que par l'interface

Il n'est possible d'accéder à une implémentation de méthode d'interface explicite que par l'interface. Ceci est illustré dans l'exemple suivant :

```
class Token: IToken, IVisitable
{
    string IToken.Name( )
    {
        ...
    }
    private void Example( )
    {
        Name( ); // Erreur de compilation

        ((IToken)this).Name( ); // OK
    }
    ...
}
```

Il est impossible de déclarer des méthodes comme étant virtuelles

Plus précisément, une classe à dériver ne peut pas accéder à une implémentation de méthode d'interface explicite et, par conséquent, aucune méthode ne peut lui être substituée. Cela implique qu'une implémentation de méthode d'interface explicite n'est pas virtuelle et qu'elle ne peut pas être déclarée comme étant virtuelle.

Il est impossible de spécifier un modificateur d'accès

Lorsque vous définissez une implémentation de méthode d'interface explicite, vous ne pouvez pas spécifier un modificateur d'accès. Cela est dû au fait que les implémentations de méthodes d'interface explicites se caractérisent par des accès différents des autres méthodes.

L'accès direct n'est pas autorisé

Une implémentation de méthode d'interface explicite n'est pas directement accessible aux clients et est donc privée dans ce sens. Ceci est illustré dans le code suivant :

```
class InOneSensePrivate
{
    void Method(Token t)
    {
        t.Name( ); // Erreur de compilation
    }
}
```

Accès indirect par des variables d'interface

Une implémentation de méthode d'interface explicite est indirectement accessible aux clients par le biais d'une variable d'interface et du polymorphisme. Dans ce sens, elle est publique. Ceci est illustré dans le code suivant :

```
class InAnotherSensePublic
{
    void Method(Token t)
    {
        ((IToken)t).Name( ); // OK
    }
}
```

Avantages d'une implémentation explicite

Les implémentations explicites de méthodes d'interface ont deux utilités principales :

1. Elles permettent aux implémentations d'interface d'être exclues de l'interface publique d'une classe ou d'un **struct**. Ceci s'avère utile lorsqu'une classe ou un **struct** implémente une interface interne qui n'est pas utile à la classe ou à l'utilisateur d'un **struct**.
2. Elles permettent à une classe ou à un **struct** de fournir différentes implémentations de méthodes d'interface qui ont la même signature. Vous trouverez ci-dessous un exemple :

```
interface IArtist
{
    void Draw( );
}
interface ICowboy
{
    void Draw( );
}
class ArtisticCowboy: IArtist, ICowboy
{
    void IArtist.Draw( )
    {
        ...
    }
    void ICowboy.Draw( )
    {
        ...
    }
}
```

Quiz : Trouver les bogues

```
interface IToken
{
    string Name( );
    int LineNumber( ) { return 42; } }
    string name;
}

class Token
{
    string IToken.Name( ) { .... }
    static void Main( )
    {
        IToken t = new IToken( );
    }
}
```

Vous pouvez travailler avec un partenaire pour trouver les bogues du code reproduit sur la diapositive. Les réponses se trouvent à la page suivante.

Réponses

Les bogues suivants se produisent dans le code de la diapositive :

1. L'interface **IToken** déclare une méthode appelée **LineNumber** qui a un corps. Une interface ne peut pas contenir d'implémentation. Le compilateur C# intercepte ce bogue comme erreur de compilation. Le code correct est le suivant :

```
interface IToken
{
    ...
    int LineNumber( );
    ...
}
```

2. L'interface **IToken** déclare un champ appelé Name. Une interface ne peut pas contenir d'implémentation. Le compilateur C# intercepte ce bogue comme erreur de compilation. Le code correct est le suivant :

```
interface IToken
{
    string Name( );
    int LineNumber( );
    //string name; // Champ commenté ici
}
```

3. La classe **Token** contient l'implémentation de la méthode d'interface explicite **IToken.Name()** mais la classe ne spécifie pas **IToken** comme interface de base. Le compilateur C# intercepte ce bogue comme erreur de compilation. Le code correct est le suivant :

```
class Token: IToken
{
    ...
}
```

4. Maintenant que **Token** spécifie **IToken** comme interface de base, elle doit implémenter les deux méthodes déclarées dans cette interface. Le compilateur C# intercepte ce bogue comme erreur de compilation. Le code correct est le suivant :

```
class Token: IToken
{
    string IToken.Name( ) { ... }
    public int LineNumber( ) { ... }
    ...
}
```


5. La méthode **Token.Main** tente de créer une instance de l'interface **IToken**. Vous ne pouvez toutefois pas créer d'instance d'une interface. Le compilateur C# intercepte ce bogue comme erreur de compilation. Le code correct est le suivant :

```
class Token: IToken
{
    ...
    static void Main( )
    {
        IToken t = new Token( );
        ...
    }
}
```

◆ Utilisation de classes abstraites (*Abstract*)

- Déclaration de classes abstraites
- Utilisation des classes abstraites dans une hiérarchie de classes
- Comparaison des classes abstraites et des interfaces
- Implémentation de méthodes abstraites
- Utilisation de méthodes abstraites
- Quiz : Trouver les bogues

Les classes abstraites sont utilisées pour fournir des implémentations de classes partielles qui peuvent être effectuées par des classes concrètes dérivées. Les classes abstraites sont particulièrement utiles pour l'implémentation partielle d'une interface qui peut ensuite être réutilisée par plusieurs classes dérivées.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- utiliser la syntaxe pour déclarer une classe abstraite ;
- expliquer comment utiliser des classes abstraites dans une hiérarchie de classes.

Déclaration de classes abstraites

■ Utilisez le mot clé **abstract**

```
abstract class Token
{
    ...
}
class Test
{
    static void Main( )
    {
        new Token( );
    }
}
```

Token
{ abstract }

Une classe abstraite ne peut
pas être instanciée

Pour déclarer une classe abstraite, utilisez le mot clé **abstract**, comme illustré sur la diapositive.

Les règles qui régissent l'utilisation d'une classe abstraite sont presque les mêmes que celles qui régissent l'utilisation d'une classe non abstraite. Les seuls points qui les différencient sont les suivants :

- Vous ne pouvez pas créer d'instance d'une classe abstraite.
Dans ce sens, les classes abstraites sont identiques aux interfaces.
- Vous pouvez créer une méthode abstraite dans une classe abstraite.
Une classe abstraite peut déclarer une méthode abstraite, mais une classe non abstraite ne le peut pas.

Les caractéristiques communes des classes abstraites et non abstraites sont les suivantes :

- Extensibilité limitée
Une classe abstraite peut étendre au plus une autre classe ou une autre classe abstraite. Notez qu'une classe abstraite peut étendre une classe non abstraite.
- Interfaces multiples
Une classe abstraite peut implémenter plusieurs interfaces.
- Méthodes d'interface héritées
Une classe abstraite doit implémenter toutes les méthodes d'interface héritées.

Utilisation de classes abstraites dans une hiérarchie de classes

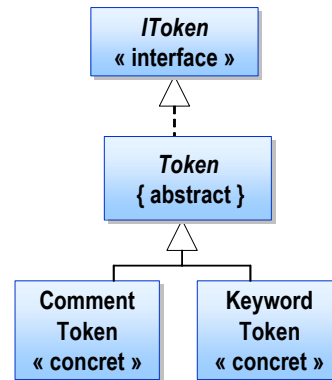
■ Exemple 1

```
interface IToken
{
    string Name( );
}

abstract class Token: IToken
{
    string IToken.Name( )
    {
        ...
    }
    ...
}

class CommentToken: Token
{
    ...
}

class KeywordToken: Token
{
    ...
}
```



Le rôle des classes abstraites dans une hiérarchie classique à trois couches constituée d'une interface, d'une classe abstraite et d'une classe concrète, consiste à fournir une implémentation totale ou partielle d'une interface.

Une classe abstraite qui implémente une interface

Consultez l'exemple 1 de la diapositive. Dans cet exemple, la classe abstraite implémente une interface. Il s'agit d'une implémentation explicite de la méthode d'interface. L'implémentation explicite n'est pas virtuelle et par conséquent ne peut pas être substituée dans des classes à dériver, telles que **CommentToken**.

Il est cependant possible pour la classe **CommentToken** de réimplémenter l'interface **IToken** comme suit :

```
interface IToken
{
    string Name( );
}

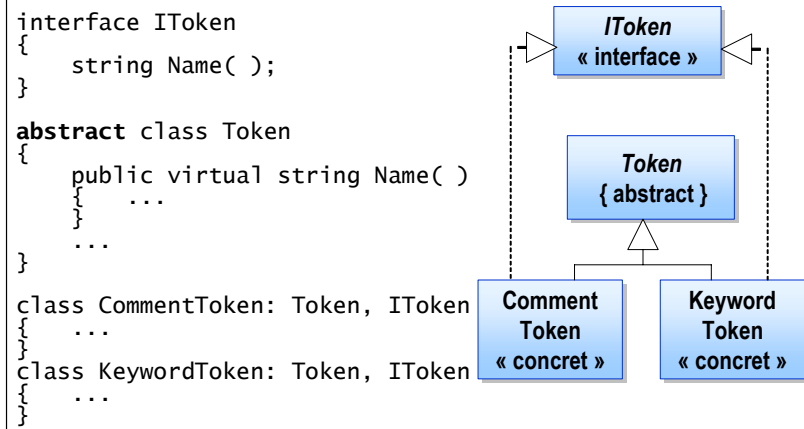
abstract class Token: IToken
{
    string IToken.Name( ) { ... }
}

class CommentToken: Token, IToken
{
    public virtual string Name( ) { ... }
}
```

Notez que dans ce cas, il n'est pas nécessaire de marquer **CommentToken.Name** comme étant une méthode **new**. Cela est dû au fait qu'une méthode de classe dérivée peut masquer uniquement une méthode de classe de base visible, mais que l'implémentation explicite de **Name** dans **Token** n'est pas directement visible dans **CommentToken**.

Utilisation de classes abstraites dans une hiérarchie de classes *(suite)*

■ Exemple 2



Un second exemple est présenté sur la diapositive pour illustrer le rôle joué par les classes abstraites dans une hiérarchie classique à trois couches.

Une classe abstraite qui n'implémente pas une interface

Consultez l'exemple 2 de la diapositive. Dans cet exemple, la classe abstraite n'implémente pas l'interface. Cela signifie que la seule façon de fournir une implémentation d'interface à une classe concrète à dériver consiste à fournir une méthode publique. La définition de la méthode dans la classe abstraite est facultativement virtuelle et elle peut donc être substituée dans les classes comme le montre le code qui suit :

```

interface IToken
{
    string Name( );
}

abstract class Token
{
    public virtual string Name( ) { ... }
}

class CommentToken: Token, IToken
{
    public override string Name( ) { ... }    // OK
}

```

Ce code montre qu'une classe peut hériter son interface et son implémentation de cette interface à partir de branches distinctes de la hiérarchie d'héritage.

Comparaison des classes abstraites et des interfaces

■ Similitudes

- Elles ne peuvent pas être instanciées
- Elles ne peuvent pas être scellées

■ Différences

- Les interfaces ne peuvent pas contenir d'implémentation
- Les interfaces ne peuvent pas déclarer de membres non publics
- Les interfaces ne peuvent étendre uniquement d'autres interfaces

Les classes abstraites et les interfaces existent toutes deux pour permettre la dérivation (ou l'implémentation). Cependant, une classe peut étendre au plus une classe abstraite ; vous devez donc être plus prudent pour la dérivation à partir d'une classe abstraite que vous ne devez l'être pour la dérivation à partir d'une interface. Réservez l'utilisation des classes abstraites pour l'implémentation de véritables relations du type « est un ».

Les similitudes entre les classes abstraites et les interfaces sont les suivantes :

- Elles ne peuvent pas être instanciées.

Cela signifie qu'on ne peut pas les utiliser directement pour créer des objets.

- Elles ne peuvent pas être scellées.

En effet, une interface scellée ne peut pas être implémentée.

Les différences entre les classes abstraites et les interfaces sont récapitulées dans le tableau ci-dessous.

Interfaces	Classes abstraites
Ne peuvent pas contenir d'implémentation	Peuvent contenir une implémentation
Ne peuvent pas déclarer de membres non publics	Peuvent déclarer des membres non publics
Peuvent étendre uniquement d'autres interfaces	Peuvent étendre d'autres classes qui peuvent être non abstraites

Lorsque vous comparez les similitudes et les différences entre les classes abstraites et les interfaces, dites-vous que les classes abstraites sont des classes non finies qui contiennent les plans de ce qui doit être fini.

Implémentation de méthodes abstraites

■ Syntaxe : Utilisez le mot clé **abstract**

```
abstract class Token
{
    public virtual string Name( ) { .... }
    public abstract int Length( );
}
class CommentToken: Token
{
    public override string Name( ) { .... }
    public override int Length( ) { .... }
}
```

■ Seules les classes abstraites peuvent déclarer des méthodes abstraites

■ Les méthodes abstraites ne peuvent pas contenir de corps de méthode

Pour déclarer une méthode abstraite, ajoutez le modificateur **abstract** à la déclaration de la méthode. La syntaxe du modificateur **abstract** est présentée sur la diapositive.

Seules les classes abstraites peuvent déclarer des méthodes abstraites. Vous trouverez ci-dessous un exemple :

```
interface IToken
{
    abstract string Name( ); // Erreur de compilation
}
class CommentToken
{
    abstract string Name( ); // Erreur de compilation
}
```

Remarque Les développeurs C++ peuvent se représenter les méthodes abstraites comme étant l'équivalent des méthodes virtuelles pures en C++.

Les méthodes abstraites ne peuvent pas contenir de corps de méthode

Les méthodes abstraites ne peuvent pas contenir d'implémentation. Ceci est illustré dans le code suivant :

```
abstract class Token
{
    public abstract string Name( ) { ... }
    // Erreur de compilation
}
```

Utilisation de méthodes abstraites

- Les méthodes abstraites sont virtuelles
- Les méthodes override peuvent se substituer aux méthodes abstraites dans des classes à dériver
- Les méthodes abstraites peuvent se substituer aux méthodes de classe de base déclarées comme virtuelles
- Les méthodes abstraites peuvent se substituer aux méthodes de classe de base déclarées comme override

Lorsque vous implémentez des méthodes abstraites, vous devez tenir compte des restrictions suivantes.

- Les méthodes abstraites sont virtuelles.
- Les méthodes override peuvent se substituer aux méthodes abstraites dans des classes à dériver.
- Les méthodes abstraites peuvent se substituer aux méthodes de classe de base qui sont déclarées comme virtuelles.
- Les méthodes abstraites peuvent se substituer aux méthodes de classe de base qui sont déclarées comme override.

Chacune de ces restrictions est décrite en détail dans les sections qui suivent.

Les méthodes abstraites sont virtuelles

Les méthodes abstraites sont considérées comme étant implicitement virtuelles mais ne peuvent pas être marquées explicitement comme étant virtuelles comme le montre le code suivant :

```
abstract class Token
{
    public virtual abstract string Name( ) { ... }
    // Erreur de compilation
}
```


Les méthodes override peuvent se substituer aux méthodes abstraites dans des classes à dériver

Étant donné que les méthodes abstraites sont virtuelles, vous pouvez les substituer dans les classes dérivées. Vous trouverez ci-dessous un exemple :

```
class CommentToken: Token
{
    public override string Name( ) {...}
}
```

Les méthodes abstraites peuvent se substituer aux méthodes de classe de base déclarées comme virtuelles

La substitution d'une méthode de classe de base déclarée comme virtuelle force une classe devant dériver à fournir sa propre implémentation de méthode et rend indisponible l'implémentation d'origine de la méthode. Vous trouverez ci-dessous un exemple :

```
class Token
{
    public virtual string Name( ) { ... }
}
abstract class Force: Token
{
    public abstract override string Name( );
}
```

Les méthodes abstraites peuvent se substituer aux méthodes de classe de base déclarées comme override

La substitution d'une méthode de classe de base déclarée comme override force une classe devant dériver à fournir sa propre implémentation de méthode et rend indisponible l'implémentation d'origine de la méthode. Vous trouverez ci-dessous un exemple :

```
class Token
{
    public virtual string Name( ) { ... }
}
class AnotherToken: Token
{
    public override string Name( ) { ... }
}
abstract class Force: AnotherToken
{
    public abstract override string Name( );
}
```

Quiz : Trouver les bogues

```
class First
{
    public abstract void Method( );
}
```

1

```
abstract class Second
{
    public abstract void Method( ) }
}
```

2

```
interface IThird
{
    void Method( );
}
abstract class Third: IThird
{
}
```

3

Vous pouvez travailler avec un partenaire pour trouver les bogues du code reproduit sur la diapositive. Les réponses se trouvent à la page suivante.

Réponses

Les bogues suivants se produisent dans le code de la diapositive :

1. Vous ne pouvez déclarer qu'une méthode abstraite dans une classe abstraite. Le compilateur C# intercepte ce bogue comme erreur de compilation. Pour corriger le code, réécrivez-le de la façon suivante :

```
abstract class First
{
    public abstract void Method( );
}
```

2. Une méthode abstraite ne peut pas déclarer un corps de méthode. Le compilateur C# intercepte ce bogue comme erreur de compilation. Pour corriger le code, réécrivez-le de la façon suivante :

```
abstract class Second
{
    public abstract void Method( );
}
```

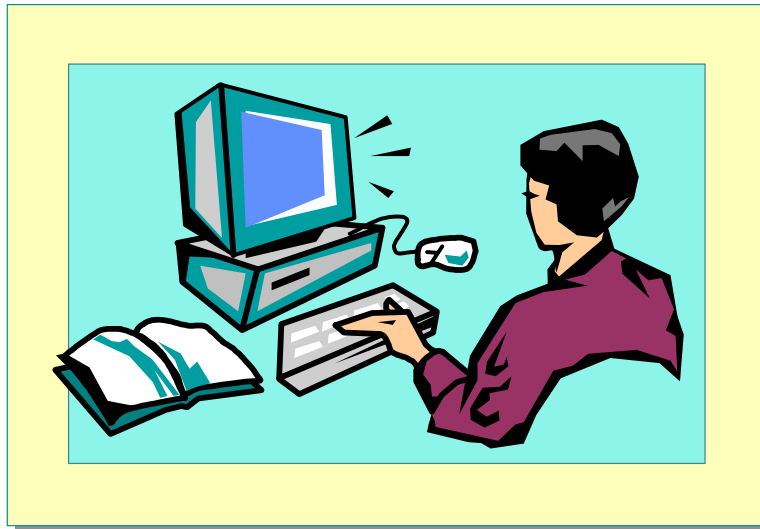
3. Le compilateur C# intercepte ce bogue comme erreur de compilation. Une classe abstraite doit fournir l'implémentation de toutes les méthodes dans les interfaces qu'elle implémente plus ou moins de la même façon qu'une classe concrète. La principale différence est que cela peut se faire directement ou indirectement lorsque vous utilisez une classe abstraite. Pour corriger le code, réécrivez-le de la façon suivante :

```
abstract class Third: IThird
{
    public virtual void Method( ) { ... }
}
```

Si vous ne voulez pas implémenter le corps de **Method** dans une classe abstraite, une autre solution consiste à déclarer la méthode comme étant abstraite pour s'assurer qu'une classe dérivée pourra l'implémenter :

```
abstract class Third: IThird
{
    public abstract void Method( );
}
```

Atelier 10.1 : Utilisation de l'héritage pour implémenter une interface



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- définir et utiliser des interfaces, des classes abstraites et des classes concrètes ;
- implémenter une interface dans une classe concrète ;
- utiliser à bon escient les mots clés **virtual** et **override** ;
- définir une classe abstraite et l'utiliser dans une hiérarchie de classes ;
- créer des classes scellées pour empêcher l'héritage.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être capables de :

- créer des classes en C# ;
- définir des méthodes pour des classes.

Durée approximative de cet atelier : 75 minutes

Exercice 1

Conversion d'un fichier source C# en fichier HTML avec coloration syntaxique

Les infrastructures sont extrêmement utiles, car elles fournissent un corps de code souple et facile à utiliser. Contrairement aux bibliothèques qui s'utilisent en appelant directement les méthodes, une infrastructure s'utilise en créant une nouvelle classe qui implémente une interface. Le code de l'infrastructure peut ensuite appeler par polymorphisme les méthodes de votre classe au moyen des opérations de l'interface. Par conséquent, un code d'infrastructure bien conçu peut être utilisé de plusieurs façons différentes, contrairement à une méthode de bibliothèque qui peut être utilisée d'une seule façon.

Scénario

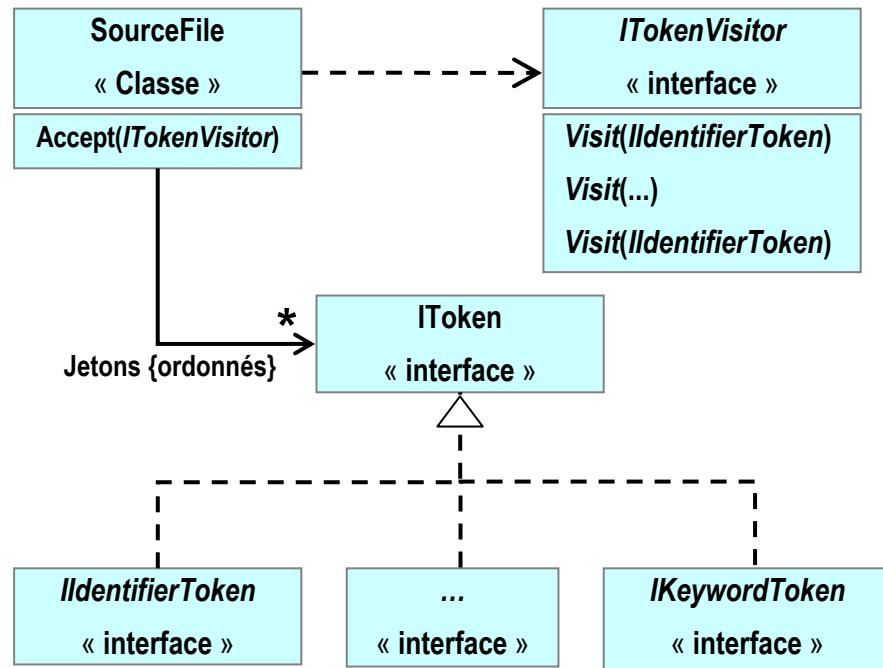
Cet exercice est basé sur une hiérarchie préécrite d'interfaces et de classes qui composent une infrastructure miniature. L'infrastructure marque à l'aide de jetons un fichier source C# et stocke les différents types de jetons dans une collection contenue dans la classe **SourceFile**. Une interface **ITokenVisitor** contenant des opérations **Visit** est également fournie laquelle, en combinaison avec la méthode **Accept** de **SourceFile**, autorise tous les jetons du fichier source à être *analysés* et à être traités en séquence. Lorsqu'un jeton est analysé, la classe peut effectuer tous les traitements requis en utilisant ce jeton.

Une classe abstraite appelée **NullTokenVisitor** a été créée pour implémenter toutes les méthodes **Visit** dans **ITokenVisitor** en utilisant des méthodes vides. Si vous ne souhaitez pas implémenter toutes les méthodes dans **ITokenVisitor**, vous pouvez dériver une classe à partir de **NullTokenVisitor** et substituer uniquement les méthodes **Visit** de votre choix.

Dans cet exercice, vous allez dériver une classe **HTMLTokenVisitor** à partir de l'interface **ITokenVisitor**. Vous allez implémenter chaque méthode **Visit** surchargée dans cette classe dérivée pour afficher sur la console le jeton présenté entre les balises HTML (Hypertext Markup Language) `` et ``. Vous exécuterez également un fichier batch simple qui lancera l'exécutable créé et redirigera le résultat affiché vers la console pour créer une page HTML utilisant une feuille de style en cascade. Vous ouvrirez ensuite la page HTML dans Microsoft Internet Explorer pour visualiser le fichier source d'origine affiché avec une coloration syntaxique.

► Pour vous familiariser avec les interfaces

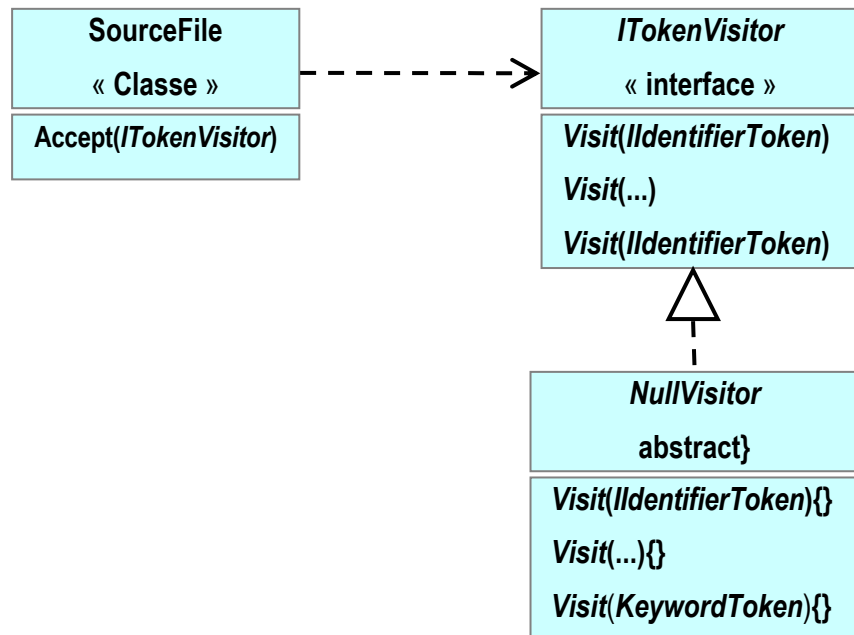
1. Ouvrez le projet ColorTokeniser.sln dans le dossier *dossier d'installation\Labs\Lab10\Starter\ColorTokeniser*.
2. Étudiez les classes et les interfaces dans les fichiers *Itoken.cs*, *Itoken_visitor.cs* et *source_file.cs*. Leur interaction s'effectue dans la hiérarchie suivante :



► Pour créer une classe abstraite **NullTokenVisitor**

1. Ouvrez le fichier `null_token_visitor.cs`.

Notez que **NullTokenVisitor** est dérivé de l'interface **ITokenVisitor**, mais qu'il n'implémente cependant aucune opération spécifiée dans l'interface. Vous allez implémenter toutes les opérations héritées de façon à ce qu'il s'agisse de méthodes vides et que **HTMLTokenVisitor** puisse être créé de façon incrémentielle.



2. Ajoutez une méthode virtuelle publique appelée **Visit** dans la classe **NullTokenVisitor**. Cette méthode retourne **void** et accepte un seul paramètre **ILineStartToken**. Le corps de la méthode sera vide. La méthode aura l'aspect suivant :

```
public class NullTokenVisitor : ITokenVisitor
{
    public virtual void Visit(ILineStartToken t) { }
    ...
}
```

3. Répétez l'étape 2 pour toutes les méthodes **Visit** surchargées déclarées dans l'interface **ITokenVisitor**.

Implémentez toutes les méthodes **Visit** dans **NullTokenVisitor** comme corps vides.

4. Enregistrez votre travail.
5. Compilez le fichier `null_token_visitor.cs`.

Si vous avez implémenté toutes les opérations **Visit** à partir de l'interface **ITokenVisitor**, la compilation doit aboutir. Si vous avez omis des opérations, le compilateur génère un message d'erreur.

6. Ajoutez une méthode statique privée void appelée **Test** dans la classe **NullTokenVisitor**.

Cette méthode n'attendra aucun paramètre. Le corps de cette méthode doit contenir une seule instruction qui crée un objet **new NullTokenVisitor**. Cette instruction vérifie que la classe **NullTokenVisitor** a implémenté toutes les opérations **Visit** et que les instances de **NullTokenVisitor** peuvent être créées. Le code de cette méthode est le suivant :

```
public class NullTokenVisitor : ITokenVisitor
{
    ...
    static void Test( )
    {
        new NullTokenVisitor( );
    }
}
```

7. Enregistrez votre travail.
8. Compilez le fichier null_token_visitor.cs et corrigez les erreurs, le cas échéant.
9. Modifiez la définition de **NullTokenVisitor**.

Étant donné que le but de la classe **NullTokenVisitor** n'est pas d'être instanciée mais d'autoriser sa dérivation, vous devez modifier sa définition de façon à la définir en tant que classe abstraite.

10. Compilez à nouveau le fichier null_token_visitor.cs.

Vérifiez également que l'instruction **new** à l'intérieur de la méthode **Test** ne génère pas d'erreur, car vous ne pouvez pas créer d'instances d'une classe abstraite.

11. Supprimez la méthode **Test**.
12. **NullTokenVisitor** doit ressembler à ce qui suit :

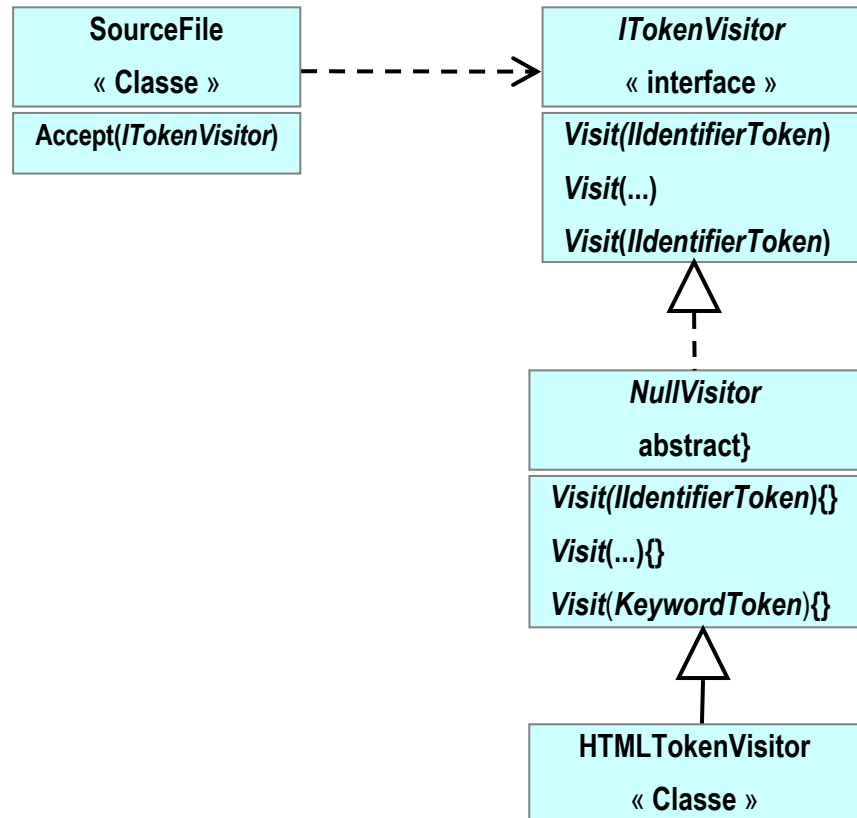
```
public abstract class NullTokenVisitor : ITokenVisitor
{
    public virtual void Visit(ILineStartToken t) { }
    public virtual void Visit(ILineEndToken t) { }

    public virtual void Visit(ICommentToken t) { }
    public virtual void Visit(IDirectiveToken t) { }
    public virtual void Visit(IIdentifierToken t) { }
    public virtual void Visit(IKeywordToken t) { }
    public virtual void Visit(IWhiteSpaceToken t) { }

    public virtual void Visit(IOtherToken t) { }
}
```


► Pour créer une classe **HTMLTokenVisitor**

1. Ouvrez le fichier `html_token_visitor.cs`.
2. Modifiez la classe **HTMLTokenVisitor** pour qu'elle dérive de la classe abstraite **NullTokenVisitor**.



3. Ouvrez le fichier `main.cs` et ajoutez deux instructions à la méthode statique **InnerMain**.
 - a. La première instruction déclare une variable appelée *visitor* du type **HTMLTokenVisitor** et l'initialise avec un nouvel objet **HTMLTokenVisitor**.
 - b. La seconde instruction passe *visitor* comme paramètre à la méthode **Accept** appelée sur la variable *source* déjà déclarée.
4. Enregistrez votre travail.
5. Compilez le programme et corrigez les erreurs, le cas échéant.

Exécutez le programme à partir de la ligne de commande en passant le nom d'un fichier source `.cs` à partir du dossier `dossier d'installation\Labs\Lab10\Starter\ColorTokeniser\bin\debug` comme argument de ligne de commande.

Rien ne se passe parce que vous n'avez encore défini aucune méthode dans la classe **HTMLTokenVisitor**.

6. Ajoutez une méthode publique non statique **Visit** à la classe **HTMLTokenVisitor**. Cette méthode retourne **void** et accepte un seul paramètre **ILineStartToken** appelé **line**.

Implémentez le corps de la méthode comme une unique instruction appelant **Write** (*et non WriteLine*) en affichant la valeur de **line.Number()** sur la console. Notez que **Number** est une opération déclarée dans l'interface **ILineStartToken**. Ne qualifiez pas la méthode avec le mot clé **virtual** ou **override**. Ceci est illustré dans le code suivant :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    public void Visit(ILineStartToken line)
    {
        Console.Write(line.Number( )); // Write et non
WriteLine
    }
}
```

7. Enregistrez votre travail.
8. Compilez le programme.

Exécutez à nouveau le programme, comme précédemment. Rien ne se passe parce que la méthode **Visit** dans **HTMLTokenVisitor** masque la méthode **Visit** dans la classe de base **NullTokenVisitor**.

9. Modifiez **HTMLTokenVisitor.Visit(ILineStartToken)** pour qu'elle se substitue à **Visit** à partir de sa classe de base.

HTMLTokenVisitor.Visit devient polymorphe, comme le montre le code qui suit :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    public override void Visit(ILineStartToken line)
    {
        Console.Write(line.Number( ));
    }
}
```

10. Enregistrez votre travail.
11. Compilez le programme et corrigez les erreurs, le cas échéant.

Exécutez le programme, comme précédemment. Le résultat est affiché. Il contient des nombres croissants sans espaces intermédiaires. (Ces nombres correspondent aux numéros de ligne générés pour le fichier que vous avez spécifié.)

12. Dans **HTMLTokenVisitor**, définissez une méthode **Visit** publique non statique et surchargée qui retourne **void** et accepte un seul paramètre **ILineEndToken**.

Cette correction ajoute une nouvelle ligne entre les lignes des jetons obtenus en sortie. Notez que cette opération est déclarée dans l'interface **ITokenVisitor**. Implémentez le corps de cette méthode pour afficher une seule nouvelle ligne sur la console, comme suit. (Notez que cette méthode utilise **WriteLine** et non **Write**.)

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    ...
    public override void Visit(ILineEndToken t)
    {
        Console.WriteLine( ); // WriteLine et non Write
    }
}
```

13. Enregistrez votre travail.

14. Compilez le programme et corrigez les erreurs, le cas échéant.

Exécutez le programme, comme précédemment. Cette fois-ci chaque numéro de ligne se termine par une ligne distincte.

► **Pour utiliser HTMLTokenVisitor et afficher les jetons d'un fichier source C#**

1. Ajoutez une méthode publique non statique **Visit** dans la classe **HTMLTokenVisitor**. Cette méthode retourne **void** et accepte un seul paramètre **IIdentifierToken** appelé **token**. Elle doit se substituer à la méthode correspondante dans la classe de base **NullTokenVisitor**.
2. Implémentez le corps de la méthode comme une instruction unique qui appelle **Write**, en affichant **token** sur la console en tant que **string**. Ceci est illustré dans le code suivant :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    ...
    public override void Visit(IIdentifierToken token)
    {
        Console.Write(token.ToString( ));
    }
}
```

Remarque Ouvrez le fichier **IToken.cs** et notez que **IIdentifierToken** est dérivé de **IToken** et que **IToken** déclare une méthode **ToString**.

3. Enregistrez votre travail.
4. Compilez le programme et corrigez les erreurs, le cas échéant.
Exécutez le programme, comme précédemment. Cette fois-ci le résultat inclut tous les identificateurs.
5. Répétez les étapes 1 à 4, en ajoutant quatre autres méthodes **Visit** surchargées dans **HTMLTokenVisitor**.
Chacune d'elle attendra un seul paramètre respectivement de type **ICommentToken**, **KeywordToken**, **IWhiteSpaceToken** et **IOtherToken**. Les corps de ces méthodes seront tous identiques à ceux décrits dans l'étape 2.

► **Pour convertir un fichier source C# en fichier HTML**

1. Le dossier *dossier d'installation*\Labs\Lab10\Starter\ColorTokeniser\bin\debug contient un script de commandes batch appelé `generate.bat`. Ce script exécute le programme `ColorTokeniser` en utilisant le paramètre de ligne de commande que vous lui avez passé. Il effectue également un pré-traitement et un post-traitement du fichier marqué par des jetons qui est produit. Il effectue ce traitement en utilisant une feuille de style en cascade (`code_style.css`) pour convertir le résultat au format HTML.
À partir de l'invite de commandes, exécutez le programme en utilisant le fichier batch `generate.bat`, et en passant le fichier `token.cs` en paramètre. (Il s'agit en fait d'une copie d'une partie du code source de votre programme mais cela fonctionnera comme un fichier `.cs` d'exemple.) Capturez le résultat dans un autre fichier avec le suffixe `.html`. Vous trouverez ci-dessous un exemple :

```
generate token.cs > token.html
```
2. Utilisez Internet Explorer pour afficher le fichier `.html` que vous venez de créer (`token.html` dans l'exemple de l'étape précédente). Vous pouvez également saisir **token.html** à l'invite.
Le résultat qui s'affiche contient plusieurs erreurs. Les numéros de ligne supérieurs à 9 sont tous indentés de façon différente par rapport aux lignes dont les numéros sont inférieurs à 10. En effet, les numéros inférieurs à 10 sont composés d'un seul chiffre tandis que les numéros supérieurs à 9 sont composés de deux chiffres. Notez également que les numéros de ligne sont de la même couleur que les jetons du fichier source, ce qui n'est pas pratique.

► **Pour corriger les erreurs liées aux numéros de ligne et à l'indentation**

1. Modifiez la définition de la méthode **Visit(ILineStartToken)** comme de façon à ajouter les éléments nécessaires pour corriger ces deux problèmes. Ceci est illustré dans le code suivant :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    public override void Visit(ILineStartToken line)
    {
        Console.Write("<span class=\"line_number\">");
        Console.Write("{0,3}", line.Number( ));
        Console.Write("</span>");
    }
    ...
}
```

2. Enregistrez votre travail.
3. Compilez le programme et corrigez les erreurs, le cas échéant.
4. Re-créez le fichier token.html à partir du fichier source token.cs à l'aide de la ligne de commande comme indiqué précédemment :

```
generate token.cs > token.html
```

5. Ouvrez le fichier token.html dans Internet Explorer.

Un problème persiste. Comparez le fichier token.html dans Internet Explorer au fichier token.cs d'origine. Notez que le premier commentaire dans token.cs (/// <summary>) apparaît comme suit dans le navigateur : « /// ». La balise <summary> n'est pas conservée. En effet, certains caractères ont une signification spécifique en HTML. Pour afficher le signe (<), le source HTML doit être **<**; et pour afficher le signe (>) le source HTML doit être **>**. Pour afficher le Et commercial (&), le source HTML doit être **&**.

► **Pour effectuer les modifications requises permettant d'afficher correctement les crochets angulaires et le Et commercial**

1. Ajoutez dans **HTMLTokenVisitor** une méthode privée non statique appelée **FilteredWrite** qui retourne **void** et attend un seul paramètre du type **IToken** appelé **token**.

Cette méthode crée une chaîne de type **string** appelée **dst** à partir de **token** et parcourt chaque caractère dans **dst**, en appliquant les modifications décrites plus haut. Le code ressemblera à ceci :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    ...
    private void FilteredWrite(IToken token)
    {
        string src = token.ToString( );
        for (int i = 0; i != src.Length; i++) {
            string dst;
            switch (src[i]) {
                case '<' :
                    dst = "&lt;"; break;
                case '>' :
                    dst = "&gt;"; break;
                case '&' :
                    dst = "&amp;"; break;
                default :
                    dst = new string(src[i], 1); break;
            }
            Console.Write(dst);
        }
    }
}
```

2. Modifiez la définition de **HTMLTokenVisitor.Visit(ICommentToken)** pour utiliser la nouvelle méthode **FilteredWrite** à la place de **Console.Write**, en procédant comme suit :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    public override void Visit(ICommentToken token)
    {
        FilteredWrite(token);
    }
    ...
}
```

3. Modifiez la définition de **HTMLTokenVisitor.Visit(IOtherToken)** pour utiliser la nouvelle méthode **FilteredWrite** à la place de **Console.Write**, en procédant comme suit :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    public override void Visit(IOtherToken token)
    {
        FilteredWrite(token);
    }
    ...
}
```

4. Enregistrez votre travail.
5. Compilez le programme et corrigez les erreurs, le cas échéant.
6. Re-créez le fichier token.html à partir du fichier source token.cs à l'aide de la ligne de commande comme indiqué précédemment :

```
generate token.cs > token.html
```

7. Ouvrez le fichier token.html dans Internet Explorer et vérifiez que les crochets angulaires et le Et commercial s'affichent correctement.

► Pour ajouter des commentaires en couleur dans le fichier HTML

1. Utilisez le Bloc-notes pour ouvrir la feuille de style code_style.css qui se trouve dans le dossier
dossier d'installation\Labs\Lab10\Starter\ColorTokeniser\bin\debug.

Le fichier de la feuille de style en cascade appelé code_style.css sera utilisé pour ajouter des couleurs dans le fichier HTML. Ce fichier a déjà été créé pour vous. Voici un exemple de son contenu :

```
...
SPAN.LINE_NUMBER
{
    background-color: white;
    color: gray;
}
...
SPAN.COMMENT
{
    color: green;
    font-style: italic;
}
```

La méthode **HTMLTokenVisitor.Visit(ILineStartToken)** utilise déjà cette feuille de style :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    public override void Visit(ILineStartToken line)
    {
        Console.Write("<span class=\"line_number\">");
        Console.Write("{0,3}", line.Number( ));
        Console.Write("</span>");
    }
    ...
}
```

Notez que cette méthode écrit « span » et « line_number » et que la feuille de style contient une entrée SPAN.LINE_NUMBER.

2. Modifiez le corps de **HTMLTokenVisitor.Visit(ICommentToken)** pour qu'elle ressemble au modèle suivant :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    public override void Visit(ICommentToken token)
    {
        Console.Write("<span class=\"comment\">");
        FilteredWrite(token);
        Console.Write("</span>");
    }
    ...
}
```

3. Enregistrez votre travail.
4. Compilez le programme et corrigez les erreurs, le cas échéant.
5. Re-créez le fichier token.html à partir du fichier source token.cs comme indiqué précédemment :

```
generate token.cs > token.html
```

6. Ouvrez le fichier token.html dans Internet Explorer.
Vérifiez que les commentaires du fichier source figurent en vert et en italique.

► **Pour ajouter des mots clés en couleur dans le fichier HTML**

1. Notez que le fichier `code_style.css` contient l'entrée suivante :

```
...  
SPAN.KEYWORD  
{  
    color: blue;  
}  
...
```

2. Modifiez le corps de **HTMLTokenVisitor.Visit(IKeywordToken)** pour utiliser le style spécifié dans la feuille de style, comme suit :

```
public class HTMLTokenVisitor : NullTokenVisitor  
{  
    public override void Visit(IKeywordToken token)  
    {  
        Console.Write("<span class=\"keyword\">");  
        FilteredWrite(token);  
        Console.Write("</span>");  
    }  
    ...  
}
```

3. Enregistrez votre travail.
4. Compilez le programme et corrigez les erreurs, le cas échéant.
5. Re-créez le fichier `token.html` à partir du fichier source `token.cs` en utilisant le fichier batch `generate.bat` comme indiqué précédemment :

```
generate token.cs > token.html
```
6. Ouvrez le fichier `token.html` dans Internet Explorer et vérifiez que les mots clés apparaissent en bleu.

► **Pour restructurer (refactor) les méthodes `Visit` et éliminer les problèmes de duplication**

1. Remarquez la duplication dans les deux méthodes **`Visit`** précédentes. Les deux méthodes affichent des chaînes de type `span` sur la console.

Vous pouvez restructurer les méthodes **`Visit`** pour éviter cette duplication.

Définissez une nouvelle méthode privée non statique appelée **`SpannedFilteredWrite`** qui retourne **`void`** et attend deux paramètres, un paramètre de type **`string`** appelé **`spanName`** et un paramètre de type **`IToken`** appelé **`token`**. Le corps de cette méthode contient trois instructions. La première instruction affiche la chaîne `span` sur la console en utilisant le paramètre **`spanName`**. La deuxième instruction appelle la méthode **`FilteredWrite`** en passant **`token`** comme argument. La troisième instruction affiche la chaîne de fin `span` sur la console. Le code ressemblera à ceci :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    ...
    private void SpannedFilteredWrite(string spanName,
    ↪IToken token)
    {
        Console.Write("<span class=\"{0}\">", spanName);
        FilteredWrite(token);
        Console.Write("</span>");
    }
    ...
}
```

2. Modifiez **`HTMLTokenVisitor.Visit(ICommentToken)`** pour utiliser cette nouvelle méthode, comme suit :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    ...
    public override void Visit(ICommentToken token)
    {
        SpannedFilteredWrite("comment", token);
    }
    ...
}
```

3. Modifiez **`HTMLTokenVisitor.Visit(IKeywordToken)`** pour utiliser cette nouvelle méthode, comme suit :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    ...
    public override void Visit(IKeywordToken token)
    {
        SpannedFilteredWrite("mot clé", token);
    }
    ...
}
```

- Modifiez le corps de la méthode

HTMLTokenVisitor.Visit(IIdentfierToken) pour qu'il appelle la méthode **SpannedFilteredWrite**. Vous devez procéder ainsi parce que les jetons des identificateurs ont également une entrée dans le fichier `code_style.css` :

```
public class HTMLTokenVisitor : NullTokenVisitor
{
    ...
    public override void Visit(IIdentfierToken token)
    {
        SpannedFilteredWrite("identfier", token);
    }
    ...
}
```

- Enregistrez votre travail.
- Compilez le programme et corrigez les erreurs, le cas échéant.
- Re-créez le fichier `token.html` à partir du fichier source `token.cs` en utilisant le fichier batch `generate.bat` comme indiqué précédemment :

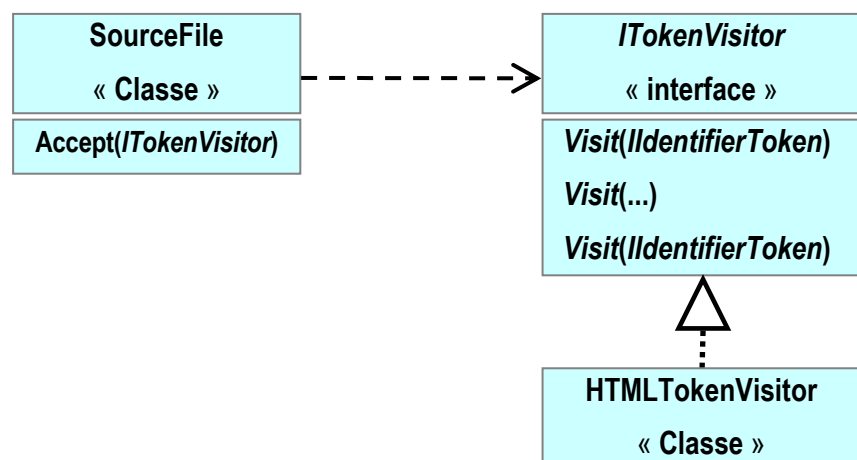
```
generate token.cs > token.html
```

- Ouvrez le fichier `token.html` dans Internet Explorer.

Vérifiez que les commentaires sont toujours en vert et que les mots clés sont toujours en bleu.

► **Pour implémenter HTMLTokenVisitor directement à partir de ITokenVisitor**

- Ouvrez le fichier `html_token_visitor.cs`.
- Modifiez le code pour que la classe **HTMLTokenVisitor** dérive de l'interface **ITokenVisitor**. Étant donné que vous avez implémenté presque toutes les méthodes **Visit** dans **HTMLTokenVisitor**, elle n'a plus besoin d'hériter de la classe abstraite **NullTokenVisitor** (qui fournit une implémentation vide par défaut de toutes les méthodes dans **ITokenVisitor**). Elle peut être dérivée directement à partir de l'interface **ITokenVisitor**.



La classe devrait ressembler à ceci :

```
public class HTMLTokenVisitor : ITokenVisitor
{
    ...
}
```

3. Enregistrez votre travail.
4. Compilez le programme.

La compilation génère de nombreuses erreurs. Le problème provient des méthodes **Visit** dans **HTMLTokenVisitor** qui sont toujours qualifiées comme étant des méthodes override. Mais vous ne pouvez pas substituer une opération dans une interface.

5. Supprimez le mot clé **override** de toutes les définitions des méthodes **Visit**.
6. Compilez le programme.

Une erreur subsiste. Le problème est dû cette fois au fait que **HTMLTokenVisitor** n'implémente pas l'opération **Visit(IDirectiveToken)** héritée de son interface **ITokenVisitor**. Auparavant, **HTMLTokenVisitor** héritait d'une implémentation vide de cette opération à partir de **NullTokenVisitor**.

7. Dans **HTMLTokenVisitor**, définissez une méthode publique non statique appelée **Visit** qui retourne **void** et attend un seul paramètre du type **IDirectiveToken** appelé *token*. Vous résoudrez ainsi le problème d'implémentation.

Le corps de cette méthode appelle la méthode **SpannedFilteredWrite** en lui passant deux paramètres : le paramètre « directive » littéral de type **string** et la variable *token*.

```
public class HTMLTokenVisitor : ITokenVisitor
{
    ...
    public void Visit(IDirectiveToken token)
    {
        SpannedFilteredWrite("directive", token);
    }
    ...
}
```

8. Enregistrez votre travail.
9. Compilez le programme et corrigez les erreurs, le cas échéant.
10. Re-créez le fichier token.html à partir du fichier source token.cs en utilisant le fichier batch generate.bat comme indiqué précédemment :

```
generate token.cs > token.html
```

11. Ouvrez le fichier token.html dans Internet Explorer.

Vérifiez que les commentaires sont toujours en vert et que les mots clés sont toujours en bleu.

► **Pour éviter l'utilisation de `HTMLTokenVisitor` comme classe de base**

1. Déclarez **`HTMLTokenVisitor`** en tant que classe scellée.

Les méthodes de **`HTMLTokenVisitor`** n'étant plus virtuelles, il est logique de déclarer **`HTMLTokenVisitor`** comme classe scellée. Ceci est illustré dans le code suivant :

```
public sealed class HTMLTokenVisitor : ITokenVisitor
{
    ...
}
```

2. Compilez le programme et corrigez les erreurs, le cas échéant.
3. Re-créez le fichier `token.html` à partir du fichier source `token.cs` en utilisant le fichier batch `generate.bat` comme indiqué précédemment :

```
generate token.cs > token.html
```

4. Ouvrez le fichier `token.html` dans Internet Explorer et vérifiez que les commentaires sont toujours en vert et les mots clés en bleu.

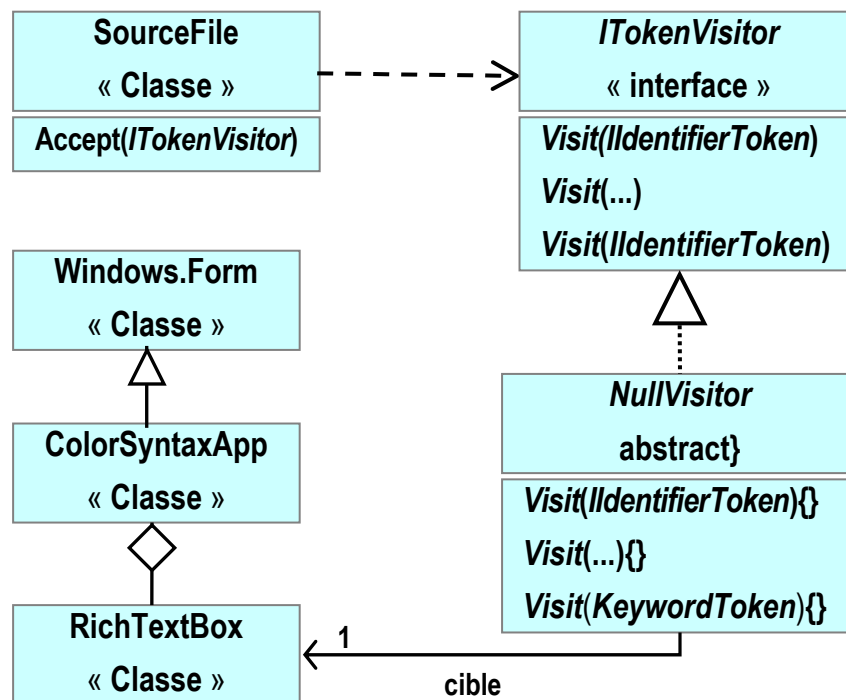
Exercice 2

Conversion d'un fichier source C# en fichier HTML avec coloration syntaxique

Dans cet exercice, vous allez étudier une deuxième application qui utilise la même infrastructure C# de générateurs de jetons que celle utilisée dans l'exercice 1.

Scénario

Dans cette application, la classe **ColorTokenVisitor** dérive de l'interface **ITokenVisitor**. Les méthodes **Visit** de cette classe écrivent des jetons en couleur dans un **RichTextBox** à l'intérieur d'une application Microsoft Windows® Forms. Les classes qui collaborent forment la hiérarchie suivante :



Pour vous familiariser avec les interfaces :

- Ouvrez le projet **ColorSyntaxApp.sln** dans le dossier *dossier d'installation\Labs\Lab10\Solution\ColourSyntaxApp*.
- Étudiez le contenu des deux fichiers .cs. Notez que la classe **ColorTokenVisitor** est très similaire à la classe **HTMLTokenVisitor** que vous avez créée dans l'exercice 1. La principale différence est que **ColorTokenVisitor** écrit les jetons en couleur dans un composant de formulaire **RichTextBox** au lieu de les afficher sur la console.
- Générez le projet.
- Exécutez l'application.
 - Cliquez sur **Ouvrir fichier**.
 - Dans la boîte de dialogue qui s'affiche, sélectionnez un fichier source .cs.
 - Cliquez sur **Ouvrir**.

Le contenu du fichier source .cs sélectionné s'affiche en couleur.

Contrôle des acquis

- Dérivation de classes
- Implémentation de méthodes
- Utilisation de classes scellées (*Sealed*)
- Utilisation d'interfaces
- Utilisation de classes abstraites (*Abstract*)

1. Créez une classe appelée **Widget** qui déclare deux méthodes publiques. Créez deux méthodes de façon à ce qu'elles retournent **void** et qu'elles n'utilisent aucun paramètre. Appelez la première méthode **First** et déclarez-la comme virtuelle. Appelez la seconde méthode **Second** et ne la déclarez pas comme virtuelle. Créez une classe appelée **FancyWidget** qui étend **Widget**, en se substituant à la méthode **First** héritée et en masquant la méthode **Second** héritée.

2. Créez une interface appelée **IWidget** qui déclare deux méthodes. Créez deux méthodes de façon à ce qu'elles retournent **void** et qu'elles n'utilisent aucun paramètre. Appelez la première méthode **First** et appelez la seconde méthode **Second**. Créez une classe appelée **Widget** qui implémente **IWidget**. Implémentez la méthode **First** comme virtuelle et implémentez la méthode **Second** explicitement.

3. Créez une classe abstraite appelée **Widget** qui déclare une méthode abstraite protégée appelée **First**, qui retourne **void** et qui n'utilise pas de paramètres. Créez une classe appelée **FancyWidget** qui étend **Widget** en se substituant à la méthode **First** héritée.

-
4. Créez une classe scellée appelée **Widget** qui implémente l'interface **IWidget** que vous avez créée à la question 2. Créez **Widget** de façon à ce qu'elle implémente les deux méthodes héritées explicitement.

Module 11 : Agrégation, espaces de noms et portée avancée

Table des matières

Vue d'ensemble	1
Utilisation de classes, de méthodes et de données internes	2
Utilisation de l'agrégation	12
Atelier 11.1 : Spécification d'accès interne	23
Utilisation des espaces de noms	29
Utilisation des modules et des assemblies	50
Atelier 11.2 : Utilisation des espaces de noms et des assemblies	58
Contrôle des acquis	64



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Utilisation de classes, de méthodes et de données internes
- Utilisation de l'agrégation
- Utilisation des espaces de noms
- Utilisation des modules et des assemblys

Dans ce module, vous apprendrez à utiliser le modificateur d'accès **internal** pour rendre le code accessible au niveau du composant ou de l'assembly. L'accès interne permet de partager l'accès aux classes et à leurs membres selon un concept similaire à celui de l'amitié en C++ et Microsoft® Visual Basic®. Vous pouvez spécifier un niveau d'accès pour un groupe de classes en collaboration, plutôt que pour une seule classe.

La création de classes individuelles bien conçues constitue l'une des règles fondamentales de la programmation orientée objet, mais quelle que soit l'ampleur du projet, vous devez créer des structures logiques et physiques à des niveaux plus élevés que celui de la classe individuelle. Vous apprendrez à regrouper des classes dans des classes plus importantes, de niveau supérieur. Vous apprendrez également à utiliser les espaces de noms pour regrouper logiquement les classes à l'intérieur d'espaces nommés, et pour créer des structures de programme logiques au-delà des classes individuelles.

Pour finir, vous apprendrez à utiliser les assemblys pour regrouper physiquement des fichiers sources collaborant entre eux dans une unité réutilisable, déployable et pouvant être associée à un numéro de version.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- utiliser l'accès interne pour autoriser des classes à avoir un accès privilégié les unes aux autres ;
- utiliser l'agrégation pour implémenter des modèles performants, tels que les fabriques ;
- utiliser les espaces de noms pour organiser les classes ;
- créer des modules et des assemblys simples.

◆ Utilisation de classes, de méthodes et de données internes

- Quelle est l'utilité de l'accès interne ?
- Accès interne
- Syntaxe
- Exemple d'accès interne

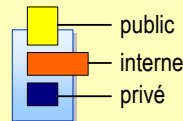
Les modificateurs d'accès définissent le niveau d'accès d'un certain code aux membres d'une classe (méthodes et propriétés par exemple). Si vous n'appliquez pas à chaque membre le modificateur d'accès voulu, le type d'accès par défaut est appliqué. Vous pouvez appliquer l'un des cinq modificateurs d'accès suivants.

Modificateur d'accès	Description
Public	Un membre déclaré public est accessible de partout. Il s'agit du modificateur d'accès le moins restrictif.
Protected	Un membre déclaré protected (protégé) n'est accessible que par les membres de la classe à laquelle il appartient et par les classes dérivées. Aucun accès de l'extérieur n'est autorisé.
Private	Un membre déclaré private (privé) n'est accessible qu'aux membres de la classe à laquelle il appartient. Même les classes dérivées ne peuvent y accéder.
Internal	Un membre déclaré internal (interne) n'est accessible qu'à l'intérieur du même assembly Microsoft .NET. En d'autres termes, il est public au niveau de l'assembly, et private pour tout membre extérieur à l'assembly.
protected internal	Un membre déclaré protected internal (interne protégé) est accessible au niveau de l'assembly en cours ou à partir de types dérivés de la classe à laquelle il appartient.

Dans cette section, vous apprendrez à utiliser l'accès interne pour définir l'accessibilité au niveau de l'assembly, plutôt qu'au niveau de la classe. Vous découvrirez l'utilité de l'accès interne, ainsi que la procédure à suivre pour déclarer des classes, des méthodes et des données internes. Pour finir, nous examinerons des exemples d'accès interne.

Quelle est l'utilité de l'accès interne ?

- Les petits objets ne sont pas utiles tout seuls
- Les objets ont besoin d'établir des relations pour constituer des objets plus grands
- Un accès plus large que l'accès à un objet individuel est nécessaire



Ajout d'objets

La création de programmes orientés objet bien conçus n'est pas simple, et la création de grands programmes orientés objet bien conçus est plus difficile encore. Il est souvent conseillé de faire en sorte que chaque entité d'un programme ne remplisse qu'un rôle, qu'elle soit petite, ciblée et facile à utiliser.

Pourtant, si vous suivez ce conseil, vous créerez un grand nombre de classes. C'est ce paradoxe qu'exprime Grady Booch quand il dit : « Si votre système est trop complexe, ajoutez-y des objets ».

Les systèmes sont complexes s'ils sont difficiles à appréhender. Or les grandes classes sont plus difficiles à appréhender que les petites classes. Par conséquent, la scission d'une grande classe en plusieurs classes plus petites permet de mieux cerner la fonctionnalité dans son ensemble.

Utilisation des relations d'objet pour créer des collaborations

Ce sont les relations entre les objets, et non pas les objets en eux-mêmes, qui confèrent sa puissance à l'orientation objet. Les objets sont construits à partir d'autres objets pour former des collaborations d'objets. Les objets qui collaborent forment eux-mêmes des entités plus larges.

Limiter l'accès à la collaboration d'objets ?

Un problème risque pourtant de se poser. Les modificateurs d'accès **public** et **private** ne s'intègrent pas parfaitement dans le modèle de collaboration des objets :

- L'accès public est illimité.
Or, vous pouvez avoir besoin de limiter l'accès aux objets de la collaboration.
- L'accès privé est limité à la classe.
Or, vous pouvez avoir besoin d'étendre l'accès aux objets de la collaboration.

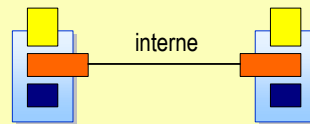
Un niveau d'accès intermédiaire, qui limiterait l'accès aux divers objets d'une collaboration donnée, serait souvent bien utile. L'accès protégé n'offre pas une solution satisfaisante, car le modificateur **protected** spécifie l'accès au niveau de la classe dans une hiérarchie d'héritage.

Dans un projet orienté objet bien conçu, les relations d'objet doivent être beaucoup plus nombreuses que les relations de classe (héritage). Vous avez donc besoin d'un mécanisme qui limite l'accès aux objets dans une collaboration donnée.

Accès interne

■ Comparaison des niveaux d'accès

- L'accès public est logique
- L'accès privé est logique
- L'accès interne est physique



Comparaison des niveaux d'accès

Il convient de différencier l'accès interne de l'accès public ou privé :

■ L'accès public est logique.

Le déploiement physique d'une classe publique (ou d'un membre d'une classe publique) n'a pas d'incidence sur son accessibilité. Quelle que soit la méthode employée pour déployer une classe publique, elle demeure publique.

■ L'accès privé est logique.

Le déploiement physique d'une classe privée (ou d'un membre d'une classe privée) n'a pas d'incidence sur son accessibilité. Quelle que soit la méthode employée pour déployer une classe privée, elle demeure privée.

■ L'accès interne est physique.

Le déploiement physique d'une classe interne (ou d'un membre d'une classe interne) a une incidence sur son accessibilité. Vous pouvez déployer une classe interne directement dans un fichier exécutable, et dans ce cas, la classe interne n'est visible que par l'unité de compilation à laquelle elle appartient. Vous pouvez aussi déployer une classe interne dans un assembly, concept que nous aborderons plus loin dans ce module. Cet assembly peut être partagé par plusieurs fichiers exécutables, mais l'accès interne n'en demeure pas moins limité à l'assembly. Si un fichier exécutable utilise plusieurs assemblies, chaque assembly possède son propre accès interne.

Comparaison entre accès interne et amitié

Dans les langages C++ et Visual Basic, le concept d'amitié permet d'autoriser les membres privés d'une classe à accéder à une autre classe. Si la classe A accorde son amitié à la classe B, les méthodes de la classe B peuvent accéder aux membres privés de la classe A. De telles amitiés créent une forte dépendance de B à A (plus forte que la dépendance qui découle de l'héritage). En effet, si la classe B était dérivée de la classe A, elle n'aurait pas accès aux membres privés de A. Pour contrebalancer cette forte dépendance, quelques restrictions de sécurité sont intégrées à l'amitié :

- L'amitié est fermée.

Si X doit accéder aux membres privés de Y, il ne peut accorder son amitié à Y. Dans ce cas, seul Y peut accorder son amitié à X.

- L'amitié n'est pas réflexive.

X peut être un ami de Y, sans que Y soit un ami de X.

L'accès interne est différent de l'amitié :

- L'accès interne est ouvert.

Vous pouvez compiler une classe C# (dans un fichier source) dans un module, puis ajouter ce module à un assembly. Ainsi, une classe peut s'accorder l'accès aux membres internes de l'assembly que d'autres classes ont rendu accessible.

- L'accès interne est réflexif.

Si X a accès aux membres internes de Y, alors Y a accès aux membres internes de X. Notez également que X et Y doivent se trouver dans le même assembly.

Syntaxe

```
internal class <ouername>
{
    internal class <nestedname> { ... }
    internal <type> field;
    internal <type> Method( ) { ... }

    protected internal class <nestedname> { ... }
    protected internal <type> field;
    protected internal <type> Method( ) { ... }
}
```

protected internal signifie **protected** ou **internal**

Si vous définissez une classe comme étant interne, vous ne pourrez y accéder qu'à partir de l'assembly en cours. Si vous définissez une classe comme étant interne protégée, vous pourrez y accéder à partir de l'assembly en cours ou de types dérivés de la classe à laquelle elle appartient.

Types non imbriqués

Vous pouvez déclarer des types directement dans la portée globale ou dans un espace de noms comme étant public ou internal, mais pas comme étant protected ou private. Le code suivant donne des exemples :

```
public class Bank { ... }    // OK
internal class Bank { ... } // OK
protected class Bank { ... } // Erreur de compilation
private class Bank { ... }  // Erreur de compilation

namespace Banking
{
    public class Bank { ... }    // OK
    internal class Bank { ... } // OK
    protected class Bank { ... } // Erreur de compilation
    private class Bank { ... }  // Erreur de compilation
}
```

Si vous déclarez des types dans la portée globale ou dans un espace de noms, et que vous ne spécifiez pas de modificateur d'accès, l'accès est implicitement interne :

```
/*internal*/ class Bank { ... }

namespace Banking
{
    /*internal*/ class Bank { ... }
    ...
}
```

Types imbriqués

Lorsque vous imbriquez des classes dans d'autres classes, vous pouvez les déclarer avec l'un des cinq types d'accessibilité, comme illustré dans le code suivant :

```
class Outer
{
    public class A { ... }
    protected class B { ... }
    protected internal class C { ... }
    internal class D { ... }
    private class E { ... }
}
```

Vous ne pouvez pas déclarer de membre d'un **struct** avec un accès `protected` ou `protected internal`, car la dérivation d'un **struct** génère une erreur de compilation. Le code suivant donne des exemples :

```
public struct S
{
    protected int x;           // Erreur de compilation
    protected internal int y; // Erreur de compilation
}
```

Conseil Lorsque vous déclarez un membre `protected internal`, l'ordre des mots clés **protected** et **internal** n'est pas significatif. Cependant, l'ordre **protected internal** est recommandé. Le code suivant fournit un exemple :

```
class BankAccount
{
    // Les deux versions sont autorisées
    protected internal BankAccount( ) { ... }
    internal protected BankAccount(decimal balance) { ... }
}
```

Remarque Les modificateurs d'accès ne peuvent pas être employés avec des destructeurs. L'exemple suivant génère une erreur de compilation :

```
class BankAccount
{
    internal ~BankAccount( ) { ... } // Erreur de compilation
}
```

Exemple d'accès interne

```
public interface IBankAccount { ... }

internal abstract class CommonBankAccount { ... }

internal class DepositAccount: CommonBankAccount,
                               IBankAccount { ... }

public class Bank
{
    public IBankAccount OpenAccount( )
    {
        return new DepositAccount( );
    }
}
```

Examinez l'exemple suivant pour apprendre à utiliser l'accès interne.

Scénario

L'exemple bancaire de la diapositive comporte trois classes et une interface. Ici, les classes et l'interface se trouvent dans un même fichier source. Elles auraient aussi bien pu être placées dans quatre fichiers sources distincts. Ces quatre types étant physiquement compilés dans un seul assembly.

L'interface **IBankAccount** et la classe **Bank** sont publiques, et elles définissent l'accessibilité à l'assembly de l'extérieur. Les classes **CommonBankAccount** et **DepositAccount** sont des classes d'implémentation uniquement, qui ne sont pas destinées à être utilisées de l'extérieur de l'assembly ; elles ne sont donc pas publiques. (Remarquez que **Bank.OpenAccount** retourne un **IBankAccount**.) Elles ne sont pourtant pas déclarées comme étant privées.

Remarquez que la classe de base abstraite **CommonBankAccount** est déclarée interne parce que le concepteur anticipe l'ajout d'autres comptes bancaires à l'assembly, et qu'il est possible que ces nouvelles classes réutilisent cette classe abstraite. Le code suivant fournit un exemple :

```
internal class CheckingAccount:
    CommonBankAccount,
    IBankAccount
{
    ...
}
```

La classe **DepositAccount** est légèrement différente. Vous pourriez également l'imbriquer dans la classe **Bank** et la rendre privée, comme suit :

```
public class Bank
{
    ...
    private class DepositAccount:
        CommonBankAccount,
        IBankAccount
    {
        ...
    }
}
```

Sur la diapositive, l'accès à la classe **DepositAccount** est déclaré interne, ce qui est moins restrictif que l'accès privé. Ce léger compromis permet plus de flexibilité dans la conception, car l'accès interne apporte :

- La séparation logique
DepositAccount peut ainsi être déclarée comme une classe non imbriquée distincte. Cette séparation logique des classes rend le code plus clair.
- La séparation physique
DepositAccount peut ainsi être placée dans son propre fichier source. Grâce à cette séparation physique, la maintenance de **DepositAccount** n'affectera aucune autre classe, et pourra être effectuée en même temps que celle des autres classes.

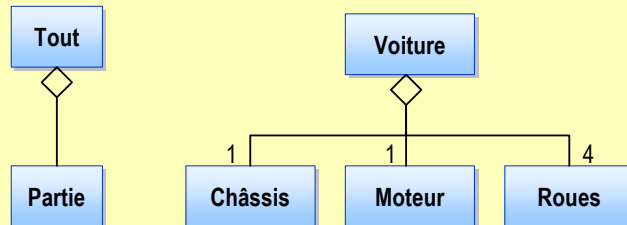
◆ Utilisation de l'agrégation

- Objets au sein d'objets
- Comparaison entre agrégation et héritage
- Fabriques
- Exemple de fabrique

Dans cette section, vous apprendrez à utiliser l'agrégation pour regrouper des objets dans le but de former une hiérarchie d'objets. L'agrégation spécifie une relation entre des objets, et non pas entre des classes. L'agrégation permet de créer des configurations d'objets réutilisables. Les configurations les plus utiles ont été documentées sous la forme de modèles. Vous apprendrez à utiliser le modèle Factory.

Objets au sein d'objets

- Les objets complexes sont construits à partir d'objets plus simples
- Les objets simples font partie d'objets plus complexes
- Cette opération est appelée *agrégation*



L'agrégation représente la relation d'objet tout/partie. La diapositive présente l'agrégation en notation UML (*Unified Modeling Language*). Le losange est placé sur la classe « tout » et une ligne relie le tout à la classe « partie ». Vous pouvez également placer sur une relation d'agrégation un numéro qui spécifie le nombre de parties dans le tout. Par exemple, la diapositive indique, en notation UML, qu'une voiture se compose d'un châssis, d'un moteur et de quatre roues. L'agrégation modélise, de manière informelle, une relation de type « a un ».

Les termes *agrégation* et *composition* sont parfois utilisés indistinctement. En UML, la composition possède une signification plus restrictive que l'agrégation :

- *Agrégation*

L'agrégation s'utilise pour spécifier une relation tout/partie dans laquelle les durées de vie du tout et des parties ne sont pas nécessairement liées, les parties peuvent être remplacées par de nouvelles et être partagées. Dans cette acception, l'agrégation est également appelée *agrégation par référence*.

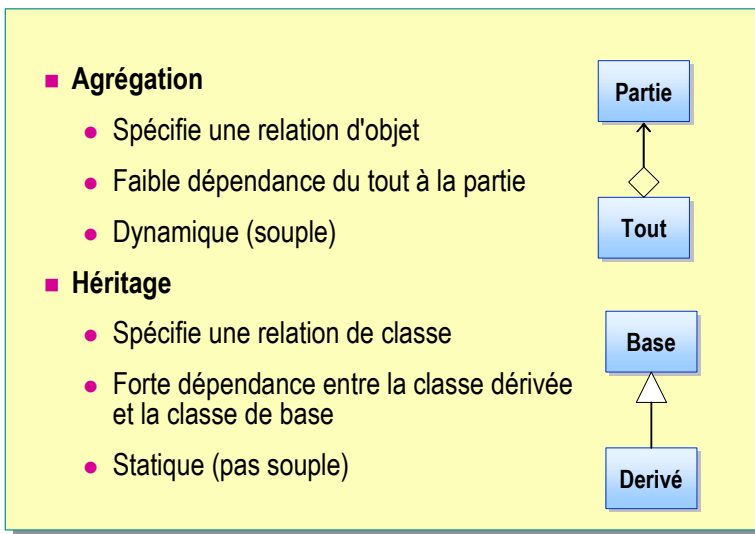
- *Composition*

La composition s'utilise pour spécifier une relation tout/partie dans laquelle les durées de vie du tout et des parties sont liées, les parties ne peuvent pas être remplacées par de nouvelles et ne peuvent pas être partagées. La composition est aussi nommée *agrégation par valeur*.

Dans une agrégation, la « classe tout » sert simplement à regrouper et à nommer les parties. En un sens, la classe tout n'existe pas vraiment. Qu'est-ce qu'une *voiture* ? C'est tout simplement le nom utilisé pour décrire une agrégation de parties spécifiques selon une configuration particulière. Mais c'est plus facile de dire *voiture* ! Dans d'autres cas, la classe tout est conceptuelle — une famille est une agrégation de personnes.

En programmation, la classe tout transmet souvent les appels de méthode à la partie appropriée. Cette opération est appelée *délégation*. Les objets agrégés forment des couches de hiérarchies de délégation. Ces hiérarchies sont parfois nommées *assemblies*, mais le terme *assembly* peut également se référer à un *assembly* physique Microsoft .NET, comme nous le verrons plus loin dans ce module.

Comparaison entre agrégation et héritage



Si l'agrégation et l'héritage permettent tous deux de créer de grandes classes à partir de petites classes, ils le font de manière totalement différentes.

Agrégation

Vous pouvez utiliser l'agrégation, qui possède les caractéristiques suivantes, pour créer de grands objets à partir de petits objets :

■ Une relation d'objet

L'agrégation spécifie une relation au niveau de l'objet. Le contrôle d'accès à la partie peut être public ou non-public. La multiplicité varie selon les objets. Un ordinateur, par exemple, est l'agrégation d'un moniteur, d'un clavier et d'une unité centrale. Certains ordinateurs possèdent toutefois deux moniteurs (pour le débogage à distance, par exemple). Certaines banques ne possèdent que quelques objets compte bancaire. D'autres en possèdent plusieurs milliers. L'agrégation peut gérer cette variation au niveau de l'objet parce que c'est une relation d'objet.

■ Faible dépendance du tout par rapport à la partie

Avec l'agrégation, les méthodes de la partie ne deviennent pas automatiquement celles du tout. Une modification de la partie ne devient pas automatiquement une modification du tout.

■ Dynamique

Le nombre de comptes bancaires contenus dans une banque peut augmenter et diminuer à mesure que les comptes bancaires sont ouverts et fermés. Si l'objet tout contient une référence à un objet partie, l'objet en référence peut être dérivé de la partie au moment de l'exécution. La référence peut même être reliée dynamiquement à des objets de plusieurs types dérivés.

L'agrégation constitue un mécanisme de structuration puissant et souple.

Héritage

L'héritage permet de créer des classes à partir de classes existantes. La relation entre la classe existante et la nouvelle classe possède les caractéristiques suivantes :

- Une relation classe

L'héritage spécifie une relation au niveau de la classe. En C#, l'héritage ne peut être que public. Il est impossible de spécifier la multiplicité d'un héritage. La *multiplicité* spécifie le nombre d'objets participant à une relation d'objet. L'héritage est déterminé au niveau de la classe. Il n'y a pas de variation au niveau de l'objet.

- Forte dépendance de la classe dérivée à la classe de base.

L'héritage crée une forte dépendance entre la classe dérivée et la classe de base. Les méthodes de la classe de base deviennent automatiquement celles de la classe dérivée. Une modification de la classe de base devient automatiquement une modification des classes dérivées.

- Statique

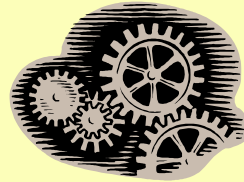
La classe de base déclarée pour une classe demeure toujours et ne peut être déclarée qu'une fois. Comparez cette caractéristique avec l'agrégation dans laquelle la référence à la partie peut être reliée dynamiquement sur des objets de diverses classes dérivées. Le type d'un objet ne peut pas être modifié. Ce manque de souplesse peut poser des problèmes. Considérons, par exemple, une hiérarchie d'héritage simple dans laquelle **Employee** est une classe de base, et **Manager** et **Programmer** sont des classes parallèles dérivées :

```
class Employee { ... }  
class Manager: Employee { ... }  
class Programmer: Employee { ... }
```

Dans cet exemple, un objet **Programmer** ne peut pas être promu objet **Manager** !

Fabriques

- La création est souvent complexe et restreinte
- De nombreux objets sont créés exclusivement dans des fabriques spécialisées
- La fabrique encapsule la création complexe
- Les fabriques sont des modèles utiles lors la modélisation de logiciels



Les nouveaux-venus dans le monde de la programmation orientée objet demandent souvent comment ils peuvent créer des constructeurs virtuels. La réponse est qu'on ne peut pas. Un constructeur de classe de base n'étant pas hérité dans une classe dérivée, il ne peut donc pas être virtuel.

Analogie

Il n'en demeure pas moins que le souhait d'abstraire les détails et la responsabilité de création se conçoit tout à fait. Cela se produit régulièrement dans la vie réelle. Vous ne pouvez pas, par exemple, créer un téléphone. C'est un processus complexe qui suppose l'acquisition et la configuration de tous les composants nécessaires à la fabrication du téléphone. Parfois la création est illégale : vous n'avez pas le droit de créer vos propres billets de banque, par exemple. Dans ces situations, la connaissance requise et la responsabilité de la création sont déléguées à un autre objet : une fabrique, dont la principale responsabilité est la création d'objets.

Encapsulation de la construction

Dans les programmes logiciels, vous pouvez aussi abstraire les détails et la responsabilité de la création en encapsulant la construction d'objets. Au lieu d'essayer de créer un constructeur virtuel dans lequel la délégation dans la hiérarchie des classes est automatique, vous pouvez déléguer manuellement dans la hiérarchie des objets :

```
class Product
{
    public void Use( ) { ... }
    ...
    internal Product( ) { ... }
}

class Factory
{
    public Product CreateProduct( )
    {
        return new Product( );
    }
    ...
}
```

Dans cet exemple, la méthode **CreateProduct** est ce que l'on appelle un modèle de méthode Factory. (Cette définition est issue de l'ouvrage *Design Patterns : Catalogue de modèles de conception réutilisables*, de E. Gamma, R. Helm, R. Johnson et J. Vlissides.) Il s'agit de la méthode d'une fabrique qui crée un produit.

Encapsulation de la destruction

L'abstraction des détails et de la responsabilité de la destruction d'un objet est également utile. Ici encore, cela se produit dans la vie réelle. Si vous ouvrez un compte dans une banque, par exemple, vous ne pouvez pas le détruire vous-même ; seule la banque peut détruire un compte. Ou encore, un citoyen respectueux de l'environnement peut, en fin de vie d'un produit, le renvoyer à l'usine qui l'a fabriqué. Son personnel pourra probablement recycler certaines parties du produit. Le code suivant fournit un exemple :

```
class Factory
{
    public Product CreateProduct( ) { ... }
    public void DestroyProduct(Product toDestroy) { ... }
    ...
}
```

Dans cet exemple, la méthode **DestroyProduct** est ce que l'on appelle une méthode Dispose, autre modèle de conception.

Expression du problème

Dans l'exemple précédent, la méthode Factory est nommée **CreateProduct**, et la méthode Dispose est nommée **DestroyProduct**. Dans une classe factory réelle, vous donnerez à ces méthodes des noms en concordance avec le vocabulaire de la fabrique. Dans une classe **Bank**, par exemple (une fabrique de comptes bancaires), vous pourriez nommer une méthode Factory **OpenAccount** (OuvrirCompte) et une méthode Dispose **CloseAccount** (FermerCompte).

Exemple de fabrique

```
public class Bank
{
    public BankAccount OpenAccount( )
    {
        BankAccount opened = new BankAccount( );
        accounts[opened.Number( )] = opened;
        return opened;
    }
    private Hashtable accounts = new Hashtable( );
}
public class BankAccount
{
    internal BankAccount( ) { ... }
    public long Number( ) { ... }
    public void Deposit(decimal amount) { ... }
}
```

Pour apprendre à utiliser le modèle Factory, nous allons examiner un exemple d'objets publics ne pouvant pas être créés, qui sont construits et agrégés dans une fabrique.

Scénario

Dans cet exemple, la classe **BankAccount** et ses méthodes sont publiques. Si vous pouviez créer un objet **BankAccount**, vous pourriez utiliser ses méthodes publiques. Cependant, vous ne pouvez pas créer d'objet **BankAccount** parce que son constructeur n'est pas public. Ce modèle est tout à fait réaliste. En effet, vous ne pouvez pas créer d'objet compte bancaire réel. Si vous voulez un compte bancaire, vous devez vous rendre dans une banque et demander à l'employé de banque d'en ouvrir un. La banque crée le compte pour vous !

C'est précisément le modèle que représente le code ci-dessus. La classe **Bank** possède une méthode publique nommée **OpenAccount**, dont le corps crée pour vous l'objet **BankAccount**. Dans cet exemple, **Bank** et **BankAccount** se trouvent dans le même fichier source ; ils feront donc partie du même assembly. (Les assemblies seront abordés plus loin dans ce module.) Si toutefois la classe **Bank** et les classes **BankAccount** étaient situées dans des fichiers sources distincts, elles pourraient être (et seraient) déployées dans le même assembly à partir duquel **Bank** aurait toujours accès au constructeur interne **BankAccount**. Remarquez aussi que **Bank** regroupe les objets **BankAccount** qu'il crée. Cette situation est très courante.

Autres solutions de conception

Pour limiter plus avant la création des objets **BankAccount**, vous pouvez définir **BankAccount** comme classe privée imbriquée de **Bank** avec une interface publique. Le code suivant fournit un exemple :

```
using System.Collections;

public interface IAccount
{
    long Number( );
    void Deposit(decimal amount);
    //...
}

public class Bank
{
    public IAccount OpenAccount( )
    {
        IAccount opened = new DepositAccount( );
        accounts[opened.Number( )] = opened;
        return opened;
    }

    private readonly Hashtable accounts = new Hashtable( );

    private sealed class DepositAccount: IAccount
    {
        public long Number( )
        {
            return number;
        }

        public void Deposit(decimal amount)
        {
            balance += amount;
        }
        //...
        // Etat de la classe
        private static long NextNumber( )
        {
            return nextNumber++;
        }
        private static long nextNumber = 123;

        // Etat de l'objet
        private decimal balance = 0.0M;
        private readonly long number = NextNumber( );
    }
}
```


Une autre solution consiste à rendre privé l'ensemble du concept **BankAccount**, et limiter l'accès au numéro de compte bancaire comme suit :

```
using System.Collections;

public sealed class Bank
{
    public long OpenAccount( )
    {
        IAccount opened = new DepositAccount( );
        long number = opened.Number( );
        accounts[number] = opened;
        return number;
    }

    public void Deposit(long accountNumber, decimal amount)
    {
        IAccount account = (IAccount)accounts[accountNumber];
        if (account != null) {
            account.Deposit(amount);
        }
    }

    //...

    public void CloseAccount(long accountNumber)
    {
        IAccount closing = (IAccount)accounts[accountNumber];
        if (closing != null) {
            accounts.Remove(accountNumber);
            closing.Dispose( );
        }
    }

    private readonly Hashtable accounts = new Hashtable( );

    private interface IAccount
    {
        long Number( );
        void Deposit(decimal amount);
        void Dispose( );
        //...
    }

    private sealed class DepositAccount: IAccount
    {
        public long Number( )
        {
            return number;
        }

        public void Deposit(decimal amount)
        {
            balance += amount;
        }
    }
}
```

Suite du code à la page suivante.

```
        public void Dispose( )
        {
            //...
            System.GC.SuppressFinalize(this);
        }

        private static long NextNumber( )
        {
            return nextNumber++;
        }
        private static long nextNumber = 123;

        private decimal balance = 0.0M;
        private readonly long number = NextNumber( );
    }
}
```

Atelier 11.1 : Spécification d'accès interne



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- spécifier l'accès interne à des classes ;
- spécifier l'accès interne à des méthodes.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez bien connaître les procédures suivantes :

- création de classes ;
- utilisation des constructeurs et des destructeurs ;
- utilisation des modificateurs d'accès **private** et **public**.

Durée approximative de cet atelier : 30 minutes

Exercice 1

Création d'une classe **Bank**

Dans cet exercice, vous effectuerez les tâches suivantes :

1. créer une classe nommée **Bank** qui agira en tant qu'emplacement de création (fabrique) des objets **BankAccount** ;
2. modifier les constructeurs **BankAccount** pour qu'ils utilisent l'accès interne ;
3. ajouter à la classe **Bank** des méthodes Factory surchargées **CreateAccount** que les clients utiliseront pour accéder aux comptes et pour demander la création de comptes ;
4. définir la classe **Bank** comme « singleton » en rendant toutes ses méthodes statiques (et publiques), et en ajoutant un constructeur privé pour empêcher la création accidentelle d'instances de la classe **Bank** ;
5. stocker des **BankAccount** dans **Bank** à l'aide d'une table de hachage **Hashtable** (**System.Collections.Hashtable**) ;
6. effectuer une simple validation pour tester la fonctionnalité de la classe **Bank**.

► Pour créer la classe **Bank**

1. Ouvrez le projet **Bank.sln** situé dans le dossier *dossier d'installation\Labs\Lab11\Exercice 1\Starter\Bank*.
2. Examinez les quatre constructeurs **BankAccount** dans le fichier **BankAccount.cs**.
Vous créerez dans la classe **Bank** quatre méthodes surchargées **CreateAccount**, qui appelleront chacune un de ces constructeurs.
3. Ouvrez le fichier **Bank.cs** et créez une méthode publique non statique de **Bank** nommée **CreateAccount**, qui n'attend aucun paramètre et qui retourne un **BankAccount**.
Le corps de cette méthode doit retourner un objet nouvellement créé **BankAccount** en appelant le constructeur **BankAccount** qui n'attend aucun paramètre.
4. Ajoutez à **Main** les instructions suivantes dans le fichier **CreateAccount.cs**.
Ce code teste votre méthode **CreateAccount**.

```
Console.WriteLine("Compte de Sid");  
Bank bank = new Bank( );  
BankAccount sids = bank.CreateAccount( );  
TestDeposit(sids);  
TestWithdraw(sids);  
Write(sids);  
sids.Dispose( );
```

5. Dans **BankAccount.cs**, faites passer l'accessibilité du constructeur **BankAccount** qui n'attend aucun paramètre de public à internal.
6. Enregistrez votre travail.
7. Compilez le programme, corrigez les erreurs le cas échéant, puis exécutez le programme.
Vérifiez si le compte bancaire de Sid a bien été créé, et si le dépôt et le retrait apparaissent dans la liste des opérations.

► Pour donner à Bank la responsabilité de la fermeture des comptes

Dans le monde réel, les comptes bancaires ne quittent jamais leur banque. Ils demeurent internes à leur banque et ce sont les clients qui accèdent à leurs comptes au moyen de leurs numéros de compte bancaire uniques. Les prochaines étapes consistent à traduire cette situation en modifiant la méthode **Bank.CreateAccount**.

1. Ajoutez à la classe **Bank** un champ privé statique nommé *accounts* de type **Hashtable**. Initialisez-le avec un objet **new Hashtable**. Vous devrez employer une *directive using* adéquate, car la classe **Hashtable** se trouve dans l'espace de noms **System.Collections**.
2. Modifiez la méthode **Bank.CreateAccount** pour qu'elle retourne le numéro **BankAccount** (type **long**), et non pas le **BankAccount** lui-même. Modifiez le corps de la méthode de telle sorte que l'objet **BankAccount** nouvellement créé soit enregistré dans les comptes **Hashtable**, avec le numéro de compte bancaire comme clé.
3. Ajoutez à la classe **Bank** une méthode publique non statique **CloseAccount**.

Cette méthode attendra un seul paramètre de type **long** (le numéro du compte à fermer) et retournera un **bool**. Le corps de cette méthode accèdera aux objets **BankAccount** par le biais des comptes **Hashtable**, avec le paramètre numéro de compte comme indexeur. Il retirera ensuite le **BankAccount** des comptes **Hashtable** en appelant la méthode **Remove** de la classe **Hashtable**, puis supprimera le compte fermé en appelant la méthode **Dispose**. La méthode **CloseAccount** retournera la valeur **true** si le paramètre de numéro de compte réussit à accéder à un **BankAccount** dans les comptes **Hashtable** ; dans le cas contraire, il retournera **false**.

À ce stade, la classe **Bank** doit se présenter comme suit :

```
using System.Collections;
```

```
public class Bank
{
    public long CreateAccount( )
    {
        BankAccount newAcc = new BankAccount( );
        long accNo = newAcc.Number( );
        accounts[accNo] = newAcc;
        return accNo;
    }
    public bool CloseAccount(long accNo)
    {
        BankAccount closing = (BankAccount)accounts[accNo];
        if (closing != null) {
            accounts.Remove(accNo);
            closing.Dispose( );
            return true;
        }
        else {
            return false;
        }
    }
    private static Hashtable accounts = new Hashtable( );
}
```

4. Enregistrez votre travail.
5. Compilez le programme.

Il ne pourra pas être compilé. Le test de validation dans **CreateAccount.Main** échoue parce que **Bank.CreateAccount** retourne un **long** à la place d'un **BankAccount**.

6. Ajoutez à la classe **Bank** une méthode publique non statique nommée **GetAccount**.

Cette méthode attend un seul paramètre de type **long** qui spécifie un numéro de compte bancaire. Elle retournera l'objet **BankAccount** enregistré dans les comptes **Hashtable**, qui possède ce numéro de compte (ou **null** si aucun compte ne porte ce numéro). L'objet **BankAccount** peut être récupéré avec le numéro de compte comme paramètre indexeur sur **accounts** (comptes), comme illustré ci-dessous :

```
public class Bank
{
    ...
    public BankAccount GetAccount(long accNo)
    {
        return (BankAccount)accounts[accNo];
    }
}
```

7. Modifiez **Main** dans le test de validation de **CreateAccount.cs** de façon à ce qu'elle utilise les nouvelles méthodes **Bank** comme suit :

```
public class CreateAccount
{
    static void Main( )
    {
        Console.WriteLine("Compte de Sid");
        Bank bank = new Bank( );
        long sidsAccNo = bank.CreateAccount( );
        BankAccount sids = bank.GetAccount(sidsAccNo);
        TestDeposit(sids);
        TestWithdraw(sids);
        Write(sids);
        if (bank.CloseAccount(sidsAccNo)) {
            Console.WriteLine("Compte fermé");
        } else {
            Console.WriteLine("Echec de la fermeture du
↪compte");
        }
    }
    ...
}
```

8. Enregistrez votre travail.
9. Compilez le programme, corrigez les erreurs le cas échéant, puis exécutez le programme. Vérifiez si le compte bancaire de Sid a bien été créé, et si le dépôt et le retrait apparaissent dans la liste des opérations.

► **Pour déclarer internes tous les constructeurs `BankAccount`**

1. Recherchez le constructeur **`BankAccount`** qui attend les paramètres de type **`AccountType`** et **`decimal`**. Remplacez son accès public par un accès interne.
2. Ajoutez une autre méthode **`CreateAccount`** à la classe **`Bank`**.
Elle sera identique à la méthode **`CreateAccount`** qui existe déjà, à cette différence près qu'elle attend deux paramètres de type **`AccountType`** et **`decimal`**, et qu'elle appellera le constructeur **`BankAccount`** qui attend ces deux paramètres.
3. Recherchez le constructeur **`BankAccount`** qui attend le paramètre unique **`AccountType`**. Remplacez son accès public par un accès interne.
4. Ajoutez une troisième méthode **`CreateAccount`** à la classe **`Bank`**.
Elle sera identique aux deux méthodes **`CreateAccount`** existantes, à cette différence près qu'elle attendra un paramètre de type **`AccountType`**, et qu'elle appellera le constructeur **`BankAccount`** qui attend ce paramètre.
5. Recherchez le constructeur **`BankAccount`** qui attend le paramètre unique **`decimal`**. Remplacez son accès public par un accès interne.
6. Ajoutez une quatrième méthode **`CreateAccount`** à la classe **`Bank`**.
Elle sera identique aux trois méthodes **`CreateAccount`** existantes, à cette différence près qu'elle attendra un paramètre de type **`decimal`**, et qu'elle appellera le constructeur **`BankAccount`** qui attend ce paramètre.
7. Enregistrez votre travail.
8. Compilez le programme et corrigez les erreurs, le cas échéant.

► Pour définir la classe **Bank** « singleton »

1. Modifiez les quatre méthodes **Bank.CreateAccount** surchargées pour les rendre statiques.
2. Modifiez la méthode **Bank.CloseAccount** pour la rendre statique.
3. Modifiez la méthode **Bank.GetAccount** pour la rendre statique.
4. Ajoutez un constructeur **Bank** privé qui empêchera la création d'objets **Bank**.
5. Modifiez **CreateAccount.Main** dans **CreateAccount.cs** de telle sorte qu'elle utilise les nouvelles méthodes statiques et qu'elle ne crée pas d'objet **Bank**, comme suit :

```
public class CreateAccount
{
    static void Main( )
    {
        Console.WriteLine("Compte de Sid");
        long sidsAccNo = Bank.CreateAccount( );
        BankAccount sids = bank.GetAccount(sidsAccNo);
        TestDeposit(sids);
        TestWithdraw(sids);
        Write(sids);
        if (Bank.CloseAccount(sidsAccNo))
            Console.WriteLine("Compte fermé");
        else
            Console.WriteLine("Echec de la fermeture du
↪compte");
    }
    ...
}
```

6. Enregistrez votre travail.
7. Compilez le programme, corrigez les erreurs le cas échéant, puis exécutez le programme. Vérifiez si le compte bancaire de Sid a bien été créé, et si le dépôt et le retrait apparaissent dans la liste des opérations.
8. Ouvrez l'invite de commandes Microsoft Visual Studio® .NET.
9. Exécutez le désassembleur ILDASM comme suit :
c:\> ildasm
10. Dans le désassembleur ILDASM, ouvrez le fichier **Bank.exe**. Notez que les quatre classes et **enum** sont listés.
11. Fermez ILDASM.
12. Fermez la fenêtre Invite de commandes Visual Studio .NET.

◆ Utilisation des espaces de noms

- Révision de la portée
- Résolution des conflits de noms
- Déclaration des espaces de noms
- Noms qualifiés complets
- Déclaration de directives d'espace de noms using
- Déclaration de directives d'alias using
- Principes d'attribution de noms pour les espaces de noms

Dans cette section, nous nous intéresserons à la portée dans le contexte des espaces de noms. Vous apprendrez à résoudre les conflits de noms par l'utilisation des espaces de noms. (Les conflits de noms se produisent lorsque deux classes ou plus dans la même portée ont le même nom.) Vous apprendrez à déclarer et à utiliser des espaces de noms. À la fin de cette section, les consignes d'utilisation des espaces de noms seront présentées.

Révision de la portée

- La portée d'un nom est la partie du programme dans laquelle vous pouvez faire référence au nom sans qualification

```
public class Bank
{
    public class Account
    {
        public void Deposit(decimal amount)
        {
            balance += amount;
        }
        private decimal balance;
    }
    public Account OpenAccount( ) { ... }
}
```

Le code de la diapositive présente quatre portées :

- La portée globale. Cette portée comprend une déclaration de membre unique : la classe **Bank**.
- La portée de la classe **Bank**. Cette portée comprend deux déclarations de membre : la classe imbriquée nommée **Account** et la méthode nommée **OpenAccount**. Notez que le type de retour de **OpenAccount** peut être simplement spécifié comme **Account**, et pas nécessairement **Bank.Account**, car **OpenAccount** est dans la même portée que **Account**.
- La portée de la classe **Account**. Cette portée comprend deux déclarations de membre : la méthode nommée **Deposit** et le champ nommé *balance*.
- Le corps de la méthode **Account.Deposit**. Cette portée contient une seule déclaration : le paramètre *amount*.

Vous ne pouvez pas utiliser sans qualification un nom qui n'est pas dans la portée. Cela se produit généralement lorsque le cadre de la portée dans laquelle le nom a été déclaré est dépassé. Cela peut également se produire si le nom est masqué. Ainsi, un membre de classe dérivée peut masquer un membre de classe de base, comme le montre le code suivant :


```
class Top
{
    public void M( ) { ... }
}
class Bottom: Top
{
    new public void M( )
    {
        M( );           // Récursivité
        base.M( );      // nécessite une qualification pour éviter
                        // la récursivité
        ...
    }
}
```

Un nom de paramètre peut masquer un nom de champ, comme suit :

```
public struct Point
{
    public Point(int x, int y)
    {
        this.x = x; // Nécessite une qualification
        this.y = y; // Nécessite une qualification
    }
    private int x, y;
}
```

Résolution des conflits de noms

- Prenons l'exemple d'un grand projet qui utilise des milliers de classes
- Que se passe-t-il si deux classes portent le même nom ?
- N'ajoutez pas de préfixes à tous les noms de classe

<pre>// De Vendor A public class Widget { ... }</pre>		<pre>public class VendorAWidget { ... }</pre>
<pre>// De Vendor B public class Widget { ... }</pre>		<pre>public class VendorBWidget { ... }</pre>

Comment gérer le problème que posent deux classes ou plus dans la même portée ayant le même nom ? En C#, les espaces de noms permettent de résoudre les conflits de noms. Les espaces de noms de C# sont similaires aux espaces de noms de C++ et aux « packages » de Java. L'accès interne ne dépend pas des espaces de noms.

Exemple d'espaces de noms

Dans l'exemple suivant, seul l'emplacement des classes dans le même assembly détermine la capacité de chaque **Method** à appeler la méthode interne **Hello** dans l'autre classe (et cela alors même que ces classes se trouvent dans des espaces de noms distincts).

```
// fichier VendorA\Widget.cs
namespace VendorA
{
    using System;

    public class Widget
    {
        internal void Hello( )
        {
            Console.WriteLine("Widget.Hello");
        }
        public void Method( )
        {
            new VendorB.ProcessMessage( ).Hello( );
        }
    }
}

// fichier VendorB\ProcessMessage.cs
namespace VendorB
{
    using System;

    public class ProcessMessage
    {
        internal void Hello( )
        {
            Console.WriteLine("ProcessMessage.Hello");
        }
        public void Method( )
        {
            new VendorA.Widget( ).Hello( );
        }
    }
}
```

Que se passe-t-il si vous n'utilisez pas d'espaces de noms ?

Si vous n'utilisez pas d'espaces de noms, vous risquez d'introduire des conflits de noms. Dans un grand projet qui comporte de nombreuses petites classes, par exemple, on peut facilement faire l'erreur de donner le même nom à deux classes.

Prenons un grand projet scindé en plusieurs sous-systèmes sur lesquels travaillent plusieurs équipes. Supposons que les sous-systèmes représentent les services organisationnels suivants :

- Services utilisateur
Moyen de permettre aux utilisateurs d'interagir avec le système.
- Services entreprise
Logique d'entreprise utilisée pour extraire, valider et manipuler les données en fonction de règles métier spécifiques.
- Services de données
Magasin de données d'un type particulier et logique de manipulation des données.

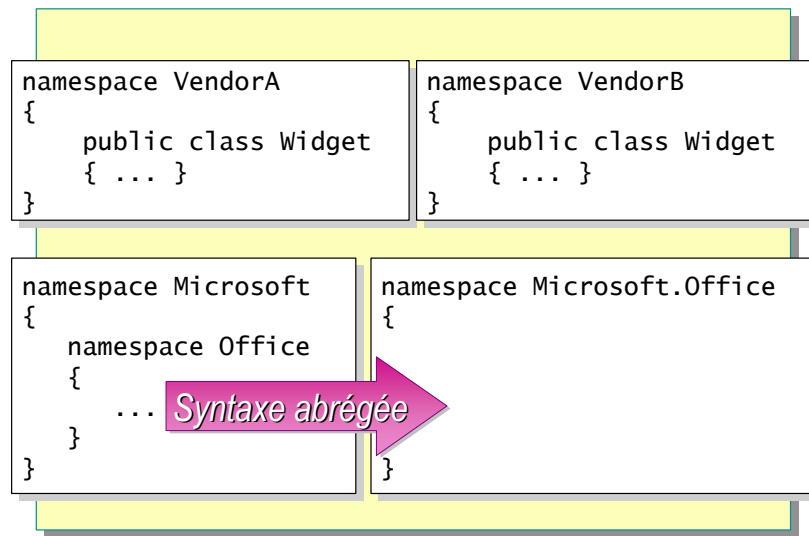
Il est fort probable que des conflits de noms se présenteront dans ce projet à plusieurs équipes. En effet, les trois équipes travaillent sur le même projet.

La solution des préfixes

Le placement d'un qualificateur de sous-système en préfixe pour chaque classe ne constitue pas une solution judicieuse pour les raisons suivantes :

- Les noms sont longs et difficiles à gérer.
Les noms de classe deviennent rapidement trop longs. Même si cela fonctionne au premier niveau de granularité, il arrive un moment où les noms de classes deviennent impossibles à gérer.
- Les noms sont complexes.
Les noms de classes sont de plus en plus difficiles à lire. Les programmes sont aussi des écrits, qui sont lus par un certain nombre de personnes. Le programme sera d'autant plus facile à maintenir qu'il sera facile à lire et à comprendre.

Déclaration des espaces de noms



Les espaces de noms permettent d'exprimer la structure logique des classes dans une syntaxe compréhensible par le compilateur.

Pour spécifier la structure des classes dans la grammaire du langage, vous devez utiliser des espaces de noms. Ainsi, au lieu d'écrire :

```
public class VendorAWidget { ... }
```

vous écrirez :

```
namespace VendorA
{
    public class Widget { ... }
}
```

Portée d'un espace de noms

Contrairement à une classe, la portée d'un espace de noms est ouverte. En d'autres termes, lorsque vous fermez un espace de noms, vous êtes autorisé à le rouvrir plus tard, et ce même dans un autre fichier source :

```
// widget.cs
namespace VendorA
{
    public class Widget { ... }
}

// ProcessMessage.cs
namespace VendorA
{
    public class ProcessMessage { ... }
}
```

Cette situation entraîne deux conséquences importantes :

- Fichiers sources multiples

Les classes en collaboration situées dans un même espace de noms peuvent être implémentées dans plusieurs fichiers sources (en général, un fichier source par classe) plutôt que dans un seul grand fichier source. Cette particularité est à comparer avec les classes imbriquées pour lesquelles la définition de toutes les classes imbriquées et de la classe externe doit se trouver dans le même fichier source.

- Espaces de noms extensibles

L'ajout d'une nouvelle classe à un espace de noms n'a pas d'incidence sur les classes qui s'y trouvent déjà. Cela contraste avec l'ajout d'une nouvelle méthode à une classe existante, qui nécessite la recompilation de toute la classe.

Espaces de noms imbriqués

Vous pouvez imbriquer deux espaces de noms pour créer une organisation à plusieurs niveaux comme suit :

```
namespace Outer
{
    namespace Inner
    {
        class Widget { ... }
    }
}
```

Ce code est assez long : il contient beaucoup d'espaces, d'accolades et son indentation le rend peu lisible. Cette syntaxe est inévitable en C++, mais elle peut être simplifiée en C# comme suit :

```
namespace Outer.Inner
{
    class Widget { ... }
}
```


Niveaux d'accès des espaces de noms

Les espaces de noms sont implicitement publics. Vous ne pouvez pas inclure de modificateur d'accès dans la déclaration d'un espace de noms, comme l'indique le code suivant :

```
namespace Microsoft.Office // OK
{
    ...
}
```

```
public namespace Microsoft.Office // Erreur de compilation
{
    ...
}
```

```
private namespace Microsoft.Office // Erreur de compilation
{
    ...
}
```

Noms qualifiés complets

- Un nom de classe qualifié complet comporte son espace de noms
- Les noms de classe non qualifiés ne peuvent être utilisés que dans la portée

```
namespace VendorA
{
    public class Widget { ... }
    ...
}
class Application
{
    static void Main( )
    {
        Widget w = new Widget( ); ❌
        VendorA.Widget w = new VendorA.Widget( ); ✅
    }
}
```

Si vous utilisez une classe à l'intérieur de son espace de noms, vous pouvez utiliser son nom abrégé, que l'on appelle *nom non qualifié*. En revanche, si vous utilisez une classe à l'extérieur de son espace de noms, elle se trouve en dehors de la portée, et vous devez utiliser son nom qualifié complet.

Noms qualifiés complets

Si vous créez une classe située dans un espace de noms, vous devez employer son nom qualifié complet lorsque vous souhaitez l'utiliser en dehors de son espace de noms. Le nom qualifié complet d'une classe comporte le nom de son espace de noms.

Dans l'exemple de la diapositive, la classe **Widget** est incorporée dans l'espace de noms **VendorA**. Vous ne pouvez donc pas utiliser le nom non qualifié **Widget** en dehors de l'espace de noms **VendorA**. La compilation du code suivant, par exemple, échouera si vous le placez dans **Application.Main** parce que **Application.Main** se trouve à l'extérieur de l'espace de noms **VendorA**.

```
Widget w = new Widget( );
```

Vous devez utiliser le nom qualifié complet de la classe **Widget** comme suit :

```
VendorA.Widget w = new VendorA.Widget( );
```

L'utilisation de noms qualifiés complets est fastidieuse et ne facilite pas la lecture du code. Nous verrons dans la rubrique suivante comment ramener les noms de classes dans la portée à l'aide des *directives using*.

Noms non qualifiés

L'utilisation des noms non qualifiés, tels que **Widget**, est limitée à la portée. La compilation du code suivant, par exemple, réussira parce que la classe **Application** a été placée dans l'espace de noms **VendorA**.

```
namespace VendorA
{
    public class Widget { ... }
}
namespace VendorA
{
    class Application
    {
        static void Main( )
        {
            Widget w = new Widget( ); // OK
        }
    }
}
```

Important Les espaces de noms permettent de regrouper logiquement des classes dans un espace nommé. Le nom de l'espace est intégré au nom complet de la classe. En revanche, il n'existe pas de relation implicite entre un espace de noms et un projet ou un assembly. Un assembly peut contenir des classes provenant de plusieurs espaces de noms, et inversement les classes d'un même espace de noms peuvent se trouver dans différents assemblies.

Déclaration de directives d'espace de noms using

■ Renvoie de façon efficace les noms dans la portée

```
namespace VendorA.SuiteB
{
    public class Widget { ... }
}

using VendorA.SuiteB;

class Application
{
    static void Main( )
    {
        Widget w = new Widget( );
    }
}
```

Les directives d'espace de noms permettent d'utiliser des classes en dehors de leurs espaces de noms, sans utiliser leurs noms qualifiés complets. Autrement dit, vous pouvez raccourcir les noms.

Utilisation des membres d'un espace de noms

Vous utilisez les *directives d'espace de noms using* pour simplifier l'utilisation des espaces de noms et des types définis dans d'autres espaces de noms. Par exemple, la compilation du code de la diapositive échouerait en l'absence de la *directive d'espace de noms using*.

```
Widget w = new Widget( );
```

Le compilateur retournerait une erreur signalant l'absence de classe globale nommée **Widget**. Grâce à la directive `using VendorA`, le compilateur est en mesure de résoudre **Widget** puisqu'il existe une classe nommée **Widget** dans l'espace de noms **VendorA**.

Espaces de noms imbriqués

Vous pouvez écrire une *directive using* avec un espace de noms imbriqué. Le code suivant fournit un exemple :

```
namespace VendorA.SuiteB
{
    public class Widget { ... }
}

//...nouveau fichier...
using VendorA.SuiteB;

class Application
{
    static void Main( )
    {
        Widget w = new Widget( );
        ...
    }
}
```

Déclaration de directives d'espace de noms de portée globale

Les *directives d'espace de noms using* de portée globale doivent apparaître avant toute déclaration de membre comme suit :

```
//...nouveau fichier...
class Widget
{
    ...
}
using VendorA;
// Déclaration après class : Erreur de compilation

//...nouveau fichier...
namespace Microsoft.Office
{
    ...
}
using VendorA;
// Déclaration après namespace : Erreur de compilation
```

Déclaration de directives *using* dans un espace de noms

Vous pouvez également déclarer une *directive using* dans un espace de noms avant toute déclaration de membre, comme suit :

```
//...nouveau fichier...
namespace Microsoft.Office
{
    using VendorA; // OK

    public class Widget { ... }
}
namespace Microsoft.PowerPoint
{
    using VendorB; // OK

    public class Widget { ... }
}
//...fin de fichier...
```

Dans ce cas, la portée de la *directive d'espace de noms using* est strictement limitée au corps de l'espace de noms dans lequel elle apparaît.

Les directives d'espace de noms *using* ne sont pas récursives

Une *directive d'espace de noms using* autorise l'accès non qualifié aux types que contient l'espace de noms spécifié, mais pas aux espaces de noms imbriqués. Par exemple, la compilation du code suivant échouera :

```
namespace Microsoft.PowerPoint
{
    public class Widget { ... }
}
namespace VendorB
{
    using Microsoft; // mais pas Microsoft.PowerPoint

    class SpecialWidget: Widget { ... }
    // Erreur de compilation
}
```

La compilation de ce code échouera parce que la *directive d'espace de noms using* fournit l'accès non qualifié aux types que contient **Microsoft**, mais pas aux espaces de noms imbriqués dans **Microsoft**. La référence à **PowerPoint.Widget** dans **SpecialWidget** est erronée, car aucun des membres nommés **PowerPoint** n'est disponible.

Noms ambigus

Examinez l'exemple suivant :

```
namespace VendorA
{
    public class Widget { ... }
}
namespace VendorB
{
    public class Widget { ... }
}
namespace Test
{
    using VendorA;
    using VendorB;

    class Application
    {
        static void Main( )
        {
            Widget w = new Widget( ); // Erreur de compilation
            ...
        }
    }
}
```

Dans ce cas, le compilateur retourne une erreur de compilation parce qu'il ne peut pas résoudre **Widget**. Le problème est le suivant : les deux espaces de noms contiennent une classe **Widget** et comportent des *directives using*. Le compilateur ne sélectionnera pas le **Widget** de **VendorA** plutôt que celui de **VendorB** parce que A précède B dans l'alphabet.

Notez toutefois que seule une tentative réelle d'utilisation du nom non qualifié **Widget** soulèvera un conflit entre les deux classes **Widget**. Vous pouvez résoudre ce problème en utilisant un nom qualifié complet pour **Widget**, l'associant de ce fait avec **VendorA** ou **VendorB**. Vous pouvez aussi réécrire le code sans utiliser le nom **Widget**, et sans provoquer d'erreur comme suit :

```
namespace Test
{
    using VendorA;
    using VendorB;

    // OK. Pas d'erreur ici.

    class Application
    {
        static void Main(string[ ] args)
        {
            VendorA.Widget w = new VendorA.Widget( );
        }
    }
}
```

Déclaration de directives d'alias using

- Crée un alias pour un espace de noms ou un type imbriqués profondément

```
namespace VendorA.SuiteB
{
    public class Widget { ... }
}
```

```
using Widget = VendorA.SuiteB.Widget;

class Application
{
    static void Main( )
    {
        Widget w = new Widget( );
    }
}
```

La *directive d'espace de noms using* place dans la portée tous les types qui se trouvent dans l'espace de noms.

Création d'alias pour les types

La *directive d'alias using* facilite l'utilisation d'un type défini dans un autre espace de noms. En l'absence de la *directive d'alias using*, la ligne de code suivante de la diapositive :

```
Widget w = new Widget( );
```

ne pourrait pas être compilée. Le compilateur retournerait une erreur signalant l'absence de classe globale nommée **Widget**. Grâce à la directive `using Widget = ...`, qui place **Widget** dans la portée, le compilateur est en mesure de résoudre ce nom. Une *directive d'alias using* ne crée jamais de type. Elle crée simplement l'alias d'un type existant. En d'autres termes, les trois instructions suivantes sont identiques :

```
Widget w = new Widget( );           // 1
VendorA.SuiteB.Widget w = new Widget( ); // 2
Widget w = new VendorA.SuiteB.Widget( ); // 3
```


Création d'alias pour les espaces de noms

Vous pouvez également recourir à la *directive d'alias using* pour faciliter l'utilisation d'un espace de noms. Par exemple, nous pouvons légèrement modifier le code de la diapositive comme suit :

```
namespace VendorA.SuiteB
{
    public class Widget { ... }
}

//... nouveau fichier ...
using Suite = VendorA.SuiteB;

class Application
{
    static void Main( )
    {
        Suite.Widget w = new Suite.Widget( );
    }
}
```

Déclaration de directives d'alias using de portée globale

Les *directives d'alias using* de portée globale doivent apparaître avant toute déclaration de membre. Le code suivant fournit un exemple :

```
//...nouveau fichier...
public class Outer
{
    public class Inner
    {
        ...
    }
}
// Déclaration après class : Erreur de compilation
using Doppelganger = Outer.Inner;
...
```

```
//...nouveau fichier...
namespace VendorA.SuiteB
{
    public class Outer
    {
        ...
    }
}
// Déclaration après namespace : Erreur de compilation
using Suite = VendorA.SuiteB;
...
```

Déclaration de directives d'alias using dans un espace de noms

Vous pouvez également placer une *directive d'alias using* dans un espace de noms avant toute déclaration de membre, comme suit :

```
//...nouveau fichier...
namespace Microsoft.Office
{
    using Suite = VendorA.SuiteB; // OK

    public class SpecialWidget: Suite.Widget { ... }
}
...
namespace Microsoft.PowerPoint
{
    using Widget = VendorA.SuiteB.Widget; // OK

    public class SpecialWidget: Widget { ... }
}
//...fin de fichier...
```

Dans ce cas, la portée de la *directive d'alias using* est strictement limitée au corps de l'espace de noms dans lequel elle apparaît. L'exemple suivant le démontre :

```
namespace N1.N2
{
    class A { }
}
namespace N3
{
    using R = N1.N2;
}
namespace N3
{
    class B: R.A { } // Erreur de compilation : R inconnu
}
```

Combinaison de directives using

L'ordre de déclaration des *directives d'espace de noms using* et des *directives d'alias using* n'importe pas. Il convient toutefois de noter que les *directives using* n'ont aucun effet les unes sur les autres ; elles s'appliquent uniquement aux déclarations de membre qui les suivent, comme l'indique l'exemple de code suivant :

```
namespace VendorA.SuiteB
{
    using System;
    using TheConsole = Console; // Erreur de compilation

    class Test
    {
        static void Main( )
        {
            Console.WriteLine("OK");
        }
    }
}
```

Ici, l'utilisation de **Console** dans **Test.Main** est autorisée parce qu'il fait partie de la déclaration de membre **Test** qui suit les *directives using*. En revanche, la *directive d'alias using* provoque une erreur de compilation parce que la *directive d'espace de noms using* qui la précède ne la concerne pas. En d'autres termes :

```
using System;
using TheConsole = Console;
```

équivalent à

```
using System;
using TheConsole = System.Console;
```

L'ordre selon lequel vous écrivez les *directives using* n'importe donc pas.

Principes d'attribution de noms pour les espaces de noms

- **Utilisez la convention PascalCasing pour séparer les composants logiques**
 - Exemple : VendorA.SuiteB
- **Ajoutez aux espaces de noms, un nom d'entreprise ou une marque connue comme préfixe**
 - Exemple : Microsoft.Office
- **Utilisez de noms pluriels le cas échéant**
 - Exemple : System.Collections
- **Évitez les conflits de noms entre espaces de noms et classes**

Voici quelques principes à suivre lors de l'attribution de noms à vos espaces de noms.

Utilisation de la convention d'affectation de noms casse Pascal (PascalCasing)

Utilisez le style d'affectation de type casse Pascal (PascalCasing) plutôt que le style de type casse mixte (camelCasing) pour les espaces de noms. Les espaces de noms sont implicitement publics. Ils suivent donc la règle générale qui s'applique aux noms publics et qui consiste à employer la notation de type casse Pascal.

Utilisation de préfixes globaux

Outre le regroupement logique, les espaces de noms permettent d'éviter les conflits de noms. Vous pouvez minimiser les risques de conflits de noms en choisissant un espace de noms unique de niveau supérieur, qui tient lieu de préfixe global. Le nom de votre entreprise ou organisation constitue un bon espace de noms de niveau supérieur. Vous pourrez, si vous le souhaitez, y imbriquer des sous-niveaux. Ainsi, un nom de projet peut constituer un espace de noms imbriqué dans l'espace de noms nom-de-l'entreprise.

Utilisation de noms pluriels le cas échéant

Si l'attribution à une classe d'un nom au pluriel est rarement justifiée, il n'en va pas de même pour un espace de noms. L'infrastructure du kit de développement Microsoft .NET Framework SDK, par exemple, comporte un espace de noms nommé **Collections** (dans l'espace de noms **System**). Le nom d'un espace de noms doit refléter son objet, qui est de regrouper des classes associées. Choisissez un nom correspondant à la fonction collective de ces classes associées. L'attribution d'un nom à un espace de noms dont les classes collaborent pour atteindre un objectif bien défini est relativement évidente.

Limitation des conflits de noms

Évitez d'attribuer le même nom à une classe et à un espace de noms. Si le code suivant est autorisé, il n'est pas recommandé :

```
namespace Wibble
{
    class Wibble
    {
        ...
    }
}
```

◆ Utilisation des modules et des assemblys

- Utilisation de modules
- Utilisation d'assemblys
- Création d'assemblys
- Comparaison entre espaces de noms et assemblys
- Utilisation du versioning

Dans cette section, vous apprendrez à créer des modules managés et des assemblys, ainsi qu'à bien différencier les espaces de noms des assemblys. Vous découvrirez également le versioning d'assemblys.

Utilisation de modules

- Les fichiers .cs peuvent être compilés dans un module managé (.netmodule).

```
csc /target:module Bank.cs
```

Création d'un module managé

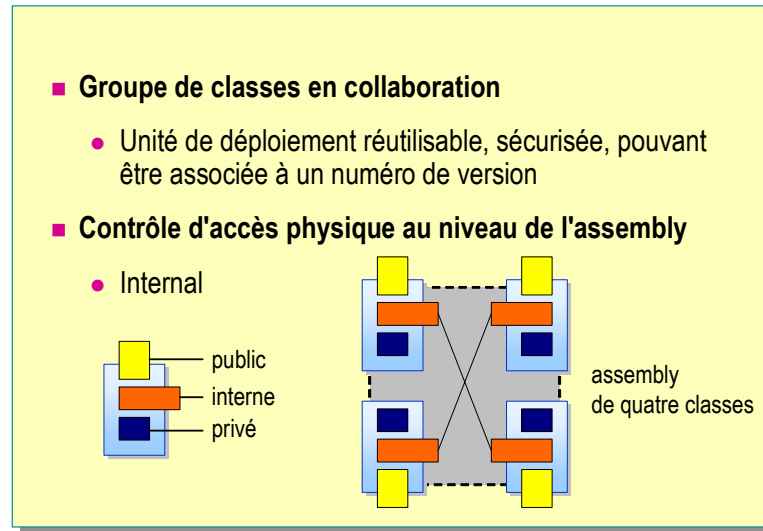
Vous pouvez compiler des fichiers sources .cs dans un module managé. Le commutateur **/target:module** du compilateur en ligne de commande CSC vous permet d'effectuer cette compilation. Les modules managés sont des versions compilées MSIL (Microsoft Intermediate Language) de fichiers de code sources qui contiennent suffisamment de métadonnées pour être auto-descriptifs. Au moment de l'exécution, le module managé est compilé plus avant par le processus JIT (just-in-time) pour le transformer en code natif de la plate-forme. Lorsqu'un module est construit à partir d'un fichier source, il reçoit par défaut l'extension .netmodule.

Pour créer un module, utilisez l'option de ligne de commande suivante :

```
c:\> csc /target:module Bank.cs
```

bank.netmodule sera le nom de ce module.

Utilisation d'assemblys



Les modules doivent être ajoutés à un assembly pour pouvoir être utilisés par un exécutable.

Qu'est ce qu'un assembly ?

Vous pouvez physiquement déployer un groupe de classes en collaboration dans un assembly, l'équivalent, en quelque sorte, d'une DLL logique. Les classes faisant partie d'un même assembly ont accès aux membres internes les unes des autres (et les classes extérieures à l'assembly n'ont pas accès à ces membres).

Un assembly est une unité de déploiement réutilisable, sécurisée, auto-descriptive et pouvant être associée à un numéro de version pour les types et les ressources ; c'est le bloc de construction principal d'une application .NET. Un assembly est constitué de deux composantes logiques : l'ensemble des types et des ressources formant une unité logique de fonctionnalité, et les métadonnées qui décrivent les liens entre ces éléments et les éléments dont ils dépendent pour fonctionner correctement. La métadonnée qui décrit un assembly s'appelle un *manifeste*. Un manifeste contient les informations suivantes :

- **Identité** : Nom textuel simple de l'assembly, numéro de version, culture facultative si l'assembly contient des ressources localisées, ainsi qu'une clé publique facultative garantissant l'unicité du nom et protégeant contre toute réutilisation non autorisée du nom.
- **Contenu** : Un assembly contient des types et des ressources. Le manifeste énumère tous les types et toutes les ressources visibles de l'extérieur de l'assembly, avec leur emplacement dans l'assembly.
- **Dépendances** : Chaque assembly décrit explicitement les autres assemblys dont il dépend, notamment la version de chaque dépendance présente au moment de la génération et du test du manifeste. Cela permet de créer une bonne configuration à laquelle revenir en cas de défaillance due à des versions non concordantes.

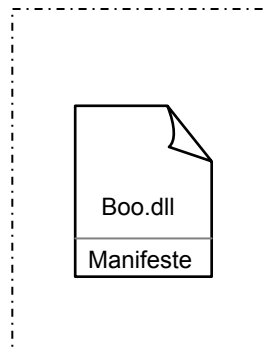
Dans le cas de figure le plus simple, un assembly se compose d'un fichier. Celui-ci contient le code, les ressources, les métadonnées des types et les métadonnées de l'assembly (manifeste). En général, les assemblies sont constitués de plusieurs fichiers. Dans ce cas, le manifeste de l'assembly peut être un fichier autonome, ou faire partie de l'un des fichiers PE contenant les types et les ressources, ou encore une combinaison des deux.

Assemblies à fichier unique et assemblies multifichiers

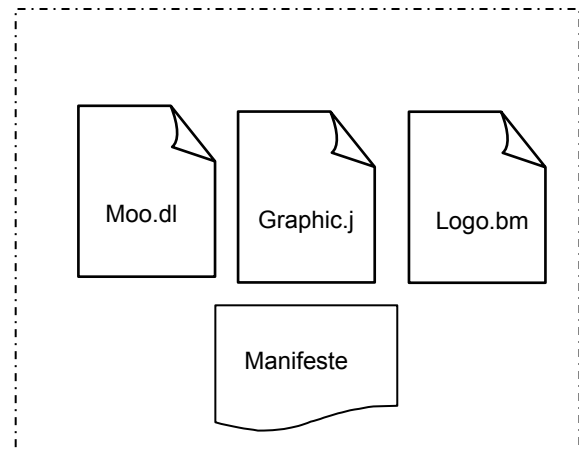
Comme leurs noms l'indiquent, un assembly à fichier unique est un assembly dont le contenu est empaqueté dans un fichier unique, tandis que le contenu d'un assembly multifichier est réparti dans plusieurs fichiers. Le manifeste peut être un fichier autonome ou son contenu peut être incorporé dans l'un des fichiers du module managé dans l'assembly.

Les assemblies à fichier unique et multifichiers sont illustrés ci-dessous.

Assembly à un fichier



Assembly multifichier



Création d'assemblies

■ Création d'un assembly à fichier

```
csc /target:library /out:Bank.dll  
    Bank.cs Account.cs
```

■ Création d'un assembly multifichier

```
csc /t:library /addmodule:Account.netmodule  
    /out:Bank.dll Bank.cs
```

Création d'un assembly à fichier unique à partir de fichiers sources

Vous pouvez créer un assembly directement à partir d'un ou de plusieurs fichiers .cs comme suit :

```
c:/> csc /target:library /out:Bank.dll Bank.cs Account.cs
```

L'outil ILDASM (Intermediate Language Disassembler) permet d'examiner les fichiers d'un assembly comme suit :

```
c:/> ildasm Bank.dll
```

Dans ce cas, l'assembly contient les types déclarés dans les fichiers .cs.

Création d'un assembly multifichier

Supposons que le type `Account` se trouve dans un fichier source distinct, et qu'il ait été compilé dans un module nommé `Account.netmodule`. Vous pouvez dans ce cas créer un assembly basé sur `Bank.cs`, et ajouter le code suivant dans le fichier module :

```
c:/> csc /t:library /addmodule:Account.netmodule /out:Bank.dll  
Bank.cs
```

L'outil ILDASM (Intermediate Language Disassembler) permet d'examiner les fichiers d'un assembly comme suit :

```
c:/> ildasm Bank.dll
```

La liste complète des options de la ligne de commande figure dans l'aide de Microsoft Visual Studio .NET.

Comparaison entre espaces de noms et assemblys

- **Espace de noms : mécanisme logique d'affectation de noms**
 - Les classes d'un espace de noms peuvent se trouver dans plusieurs assemblys
 - Les classes de plusieurs espaces de noms peuvent se trouver dans un même assembly
- **Assembly : mécanisme physique de regroupement**
 - Le MSIL de l'assembly et le manifeste y sont contenus directement
 - Les modules de l'assembly et les ressources peuvent être des liens externes

Un espace de noms est un mécanisme logique de compilation. Il fournit une structure logique aux noms des entités du code source. Les espaces de noms ne sont pas des entités d'exécution.

Un assembly est un mécanisme physique d'exécution. Il fournit une structure physique aux composants d'exécution qui constituent l'exécutable.

Comparaison entre espaces de noms et assemblys

Les classes regroupées dans un même espace de noms peuvent être déployées dans plusieurs assemblys. Les classes regroupées dans plusieurs espaces de noms peuvent être déployées dans un seul assembly. Il est toutefois judicieux de conserver une correspondance logique-physique aussi étroite que possible.

Les espaces de noms et les assemblys sont similaires en ce qui concerne l'emplacement physique de leurs éléments :

- les éléments d'un namespace n'ont pas besoin de résider physiquement dans un seul fichier source ; les éléments d'un namespace peuvent (et doivent en principe) être maintenus dans des fichiers sources distincts ;
- les éléments référencés par un assembly n'ont pas besoin de résider dans l'assembly ; vous avez vu précédemment que l'assembly ne renferme pas physiquement les modules compris dans un espace de noms ; l'assembly contient un lien nommé au module externe.

Utilisation du versioning

- Chaque assembly dispose d'un numéro de version qui fait partie de son identité
- Le numéro de version est représenté physiquement en tant que membre en quatre parties au format suivant :
<version majeure>.<version mineure>.<numéro de compilation>.<révision>

Chaque assembly possède un numéro de version de compatibilité spécifique qui fait partie de son identité. Deux assemblies portant des numéros de versions différents constituent deux assemblies distincts du point de vue du chargeur de classe du CLR (Common Language Runtime).

Format du numéro de version

Le numéro de version de compatibilité est représenté par un numéro constitué de quatre parties :

<version majeure>.<version mineure>.<numéro de compilation>.<révision>

Par exemple, dans le numéro de version 1.5.1254.0 : 1 représente la version majeure, 5 la version mineure, 1254 le numéro de compilation, et 0 le numéro de révision.

Le runtime ne contrôle pas la compatibilité de version dans les références aux assemblies privés. En effet, les assemblies privés sont déployés avec l'application et hébergés dans le même répertoire ou sous-répertoire que l'application ; l'auteur maîtrise ainsi complètement les contenus et la distribution de ces assemblies. En outre, les assemblies privés visent uniquement l'application avec laquelle ils sont déployés.

Les assemblies partagés et les fichiers de configuration dépassent le cadre de ce cours.

Atelier 11.2 : Utilisation des espaces de noms et des assemblys



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- utiliser l'agrégation pour regrouper des objets dans une hiérarchie ;
- organiser des classes dans des espaces de noms.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez bien connaître les procédures suivantes :

- création de classes ;
- utilisation des constructeurs et des destructeurs ;
- utilisation des modificateurs d'accès **private** et **public**.

Durée approximative de cet atelier : 30 minutes

Exercice 1

Organisation de classes

Dans cet exercice, vous organiserez des classes dans un espace de noms **Banking**, et vous créerez et référencerez un assembly. Pour ce faire, vous effectuerez les procédures suivantes :

1. Placement de l'enum **AccountType** et des classes **Bank**, **BankAccount** et **BankTransaction** dans l'espace de noms **Banking**, puis compilation dans une bibliothèque.
2. Modification du test de validation. Au départ, le test se référera aux classes par leurs noms qualifiés complets. Vous le modifierez avec une directive *using*.
3. Compilation du test de validation dans un assembly qui référence la bibliothèque **Banking**.
4. Utilisation de l'outil ILDASM pour vérifier que le test de validation .exe se réfère à la DLL Banking, et ne contient pas les classes **Bank** et **BankAccount**.

► Pour placer toutes les classes dans l'espace de noms Banking

1. Ouvrez le projet Bank.sln situé dans *dossier d'installation\Labs\Lab11\Exercice 2\Starter\Bank*.
2. Modifiez **AccountType enum** dans AccountType.cs pour l'imbriquer dans l'espace de noms **Banking** comme suit :

```
namespace Banking
{
    public enum AccountType { ... }
}
```

3. Modifiez la classe **Bank** dans Bank.cs pour l'imbriquer dans l'espace de noms **Banking** comme suit :

```
namespace Banking
{
    public class Bank
    {
        ...
    }
}
```

4. Modifiez la classe **BankAccount** dans BankAccount.cs pour l'imbriquer dans l'espace de noms **Banking** comme suit :

```
namespace Banking
{
    sealed public class BankAccount
    {
        ...
    }
}
```

5. Modifiez la classe **BankTransaction** dans BankTransaction.cs pour l'imbriquer dans l'espace de noms **Banking** comme suit :

```
namespace Banking
{
    public class BankTransaction
    {
        ...
    }
}
```

6. Enregistrez votre travail.
7. Compilez le programme. La compilation échoue. Le compilateur est incapable de résoudre les références à **Bank**, **BankAccount** et **BankTransaction** dans le fichier CreateAccount.cs parce que ces classes se trouvent actuellement dans l'espace de noms **Banking**. Modifiez **CreateAccount.Main** pour expliciter ces références. Par exemple,

```
static void write(BankAccount acc) { ... }
```

doit indiquer :

```
static void write(Banking.BankAccount acc) { ... }
```

8. Enregistrez votre travail.
9. Compilez le programme et corrigez les erreurs, le cas échéant. Vérifiez si le compte bancaire de Sid a bien été créé, et si le dépôt et le retrait apparaissent dans la liste des opérations.
10. Ouvrez la ligne de commande Visual Studio .NET.
11. Exécutez ILDASM.
12. Ouvrez Bank.exe dans ILDASM. Ce fichier se trouve dans *dossier d'installation*\Labs\Lab11\Exercise2\Starter\Bank\bin\debug.
13. Constatez que les trois classes et l'**enum** figurent à présent dans l'espace de noms **Banking**, et que la classe **CreateAccount** est présente.
14. Fermez ILDASM.

► **Pour créer et utiliser une bibliothèque Banking**

1. Ouvrez une ligne de commande Visual Studio .NET et accédez à *dossier d'installation*\Labs\Lab11\Exercise2\Starter\Bank. À partir de la ligne de commande, créez la bibliothèque banking comme suit :

```
c:\> csc /target:library /out:bank.dll a*.cs b*.cs  
c:\> dir  
...  
bank.dll  
...
```
2. À partir de la ligne de commande, exécutez ILDASM en passant la DLL en paramètre comme suit :

```
c:\> ildasm bank.dll
```
3. Constatez que les trois classes « Bank* » et l'**enum** figurent toujours dans l'espace de noms **Banking**, mais que la classe **CreateAccount** n'y figure plus. Fermez ILDASM.
4. À partir de la ligne de commande, compilez le test de validation que renferme CreateAccount.cs dans un assembly qui référence la bibliothèque **Banking** bank.dll comme suit :

```
c:\> csc /reference:bank.dll createaccount.cs  
c:\> dir  
...  
createaccount.exe  
...
```
5. À partir de la ligne de commande, exécutez ILDASM en indiquant le nom de l'exécutable en paramètre comme suit :

```
c:\> ildasm createaccount.exe
```
6. Constatez que les quatre classes et l'**enum** ne font plus partie de createaccount.exe. Double-cliquez sur l'élément MANIFEST dans ILDASM pour ouvrir la fenêtre **MANIFEST**. Examinez le manifeste. Remarquez que l'exécutable fait référence à la bibliothèque Banking, mais ne la contient pas :

```
.assembly extern bank
```
7. Fermez ILDASM.

► **Pour simplifier le test de validation à l'aide d'une directive using**

1. Editez CreateAccount.cs et supprimez toutes les occurrences de l'espace de noms **Banking**. Par exemple,

```
static void write(Banking.BankAccount acc) { ... }
```

doit indiquer :

```
static void write(BankAccount acc) { ... }
```

2. Enregistrez votre travail.
3. Essayez de compiler le programme. La compilation échoue. **Bank**, **BankAccount** et **BankTransaction** demeurent introuvables.
4. Ajoutez au début de CreateAccount.cs une *directive using* qui utilise **Banking**, comme suit :
using Banking;
5. Compilez le programme, corrigez les erreurs le cas échéant, puis exécutez le programme. Vérifiez si le compte bancaire de Sid a bien été créé, et si le dépôt et le retrait apparaissent dans la liste des opérations.

► Pour étudier plus avant les méthodes internes

1. Éditez la méthode **Main** dans le test de validation `CreateAccount.cs`. Ajoutez-y une instruction qui crée un objet **BankTransaction** comme suit :

```
static void Main( )
{
    new BankTransaction(0.0M);
    ...
}
```

2. Enregistrez votre travail.
3. Ouvrez une invite de commandes Visual Studio .NET et accédez à *dossier d'installation*\Labs\Lab11\Exercise2\Starter\Bank. À partir de la ligne de commande, entrez la ligne de code suivante pour vérifier si vous pouvez créer un exécutable qui *n'utilise pas* la bibliothèque `banking` :

```
c:\> csc /out:createaccount.exe *.cs
```

4. À partir de la ligne de commande, vérifiez si vous pouvez créer un exécutable qui *utilise* la bibliothèque `banking` :

```
c:\> csc /target:library /out:bank.dll a*.cs b*.cs
c:\> csc /reference:bank.dll createaccount.cs
```

5. Dans un cas comme dans l'autre, l'instruction supplémentaire dans **Main** ne pose pas de problème. En effet, le constructeur **BankTransaction** dans `BankTransaction.cs` est public.
6. Modifiez la classe **BankTransaction** dans `BankTransaction.cs` pour spécifier l'accès interne de son constructeur.
7. Enregistrez votre travail.
8. À partir de la ligne de commande, vérifiez si vous pouvez *toujours* créer un exécutable qui *n'utilise pas* la bibliothèque `banking` :

```
c:\> csc /out:createaccount.exe *.cs
```

9. À partir de la ligne de commande, vérifiez si vous ne pouvez pas créer d'exécutable qui *utilise* la bibliothèque `banking` :

```
c:\> csc /target:library /out:bank.dll a*.cs b*.cs
c:\> csc /reference:bank.dll createaccount.cs
....erreur CS0122:
'Banking.BankTransaction.BankTransaction(decimal)' est
inaccessible en raison de son niveau de protection
```

10. Dans **CreateAccount.Main**, supprimez l'instruction qui crée un objet **BankTransaction**.
11. Enregistrez votre travail.
12. Vérifiez à nouveau si vous pouvez compiler le test de validation dans un assembly qui référence la bibliothèque **Banking**.

```
c:\> csc /target:library /out:bank.dll a*.cs b*.cs
c:\> csc /reference:bank.dll createaccount.cs
```

Contrôle des acquis

- Utilisation de classes, de méthodes et de données internes
- Utilisation de l'agrégation
- Utilisation des espaces de noms
- Utilisation des modules et des assemblys

-
1. Supposons que nous avons deux fichiers .cs. Le fichier alpha.cs contient une classe nommée **Alpha** qui comporte une méthode interne nommée **Method**. Le fichier beta.cs contient une classe nommée **Beta** qui comporte aussi une méthode interne nommée **Method**. Est-ce que **Alpha.Method** peut être appelée à partir de **Beta.Method**, et inversement ?
 2. L'agrégation est-elle un type de relation d'objet ou un type de relation de classe ?

3. Le code ci-dessous sera-t-il compilé sans erreur ?

```
namespace Outer.Inner
{
    class Wibble { }
}
namespace Test
{
    using Outer.Inner;
    class SpecialWibble: Inner.Wibble { }
}
```

4. Un programme exécutable peut-il référencer directement un module managé ?

Module 12 : Opérateurs, délégués et événements

Table des matières

Vue d'ensemble	1
Introduction aux opérateurs	2
Surcharge d'opérateurs	6
Atelier 12.1 : Définition d'opérateurs	19
Création et utilisation de délégués	37
Définition et utilisation d'événements	47
Démonstration : Gestion d'événements	53
Atelier 12.2 : Définition et utilisation d'événements	54
Contrôle des acquis	63



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

Vue d'ensemble

- Introduction aux opérateurs
- Surcharge d'opérateurs
- Création et utilisation de délégués
- Définition et utilisation d'événements

Ce module aborde trois types de fonctionnalités utiles : les opérateurs, les délégués et les événements.

Les opérateurs sont les composants de base d'un langage. Vous pouvez les utiliser pour effectuer des manipulations et des comparaisons entre des variables qui peuvent être logiques, relationnelles ou conditionnelles par nature.

Les délégués spécifient un contrat entre un objet qui émet des appels à une fonction et un objet qui implémente la fonction appelée.

Les événements fournissent à une classe le moyen d'avertir ses clients en cas de modification de l'état d'un de ses objets.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- définir des opérateurs afin de simplifier l'utilisation d'une classe ou d'un **struct** ;
- utiliser des délégués pour dissocier l'appel à une méthode de l'implémentation de cette méthode ;
- ajouter des spécifications d'événements à une classe pour permettre d'avertir les classes abonnées des changements d'état de l'objet.

◆ Introduction aux opérateurs

- Opérateurs et méthodes
- Opérateurs C# prédéfinis

Les opérateurs sont différents des méthodes. Ils présentent des exigences spéciales qui leur permettent de fonctionner de la manière escomptée. C# possède une série d'opérateurs prédéfinis que vous pouvez utiliser pour manipuler les types et classes fournis avec Microsoft® .NET Framework.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- identifier pourquoi C#, comme la plupart des langages, possède des opérateurs ;
- définir des opérateurs afin de rendre une classe ou un **struct** plus facile à utiliser.

Opérateurs et méthodes

■ Utilisation de méthodes

- Clarté réduite
- Risque accru d'erreurs syntaxiques et sémantiques

```
myIntVar1 = Int.Add(myIntVar2,
                    Int.Add(Int.Add(myIntVar3,
                                    myIntVar4), 33));
```

■ Utilisation d'opérateurs

- Clarté des expressions

```
myIntVar1 = myIntVar2 + myIntVar3 + myIntVar4 + 33;
```

L'objectif des opérateurs consiste à rendre les expressions claires et faciles à comprendre. Il pourrait exister un langage sans opérateur, se basant plutôt sur des méthodes bien définies, mais cela aurait probablement un effet négatif sur sa clarté.

Utilisation de méthodes

Supposons, par exemple, qu'il n'y ait pas d'opérateur d'addition arithmétique et que le langage fournisse plutôt une méthode **Add** de la classe **Int** qui accepte des paramètres et retourne un résultat. Dans ce cas, pour ajouter deux variables, vous devriez écrire un code ressemblant à ce qui suit :

```
myIntVar1 = Int.Add(myIntVar2, myIntVar3);
myIntVar2 = Int.Add(myIntVar2, 1);
```

Utilisation d'opérateurs

À l'aide de l'opérateur d'addition arithmétique, vous pouvez écrire des lignes de code plus concises, comme ci-dessous :

```
myIntVar1 = myIntVar2 + myIntVar3;
myIntVar2 = myIntVar2 + 1;
```

Le code devient presque indéchiffrable si vous additionnez une série de valeurs à l'aide de la méthode **Add**, comme dans le code suivant :

```
myIntVar1 = Int.Add(myIntVar2, Int.Add(Int.Add(myIntVar3,
                                              myIntVar4), 33));
```

Si vous utilisez des méthodes de cette manière, la probabilité d'erreurs (syntaxiques et sémantiques) est considérable. En réalité, les opérateurs sont implémentés comme méthodes par C#, mais leur syntaxe est conçue pour les rendre conviviaux. Le compilateur et le runtime C# convertissent automatiquement les expressions contenant des opérateurs en séries correctes d'appels de méthode.

Opérateurs C# prédéfinis

Catégories d'opérateurs	
Arithmétique	Accès au membre
Logique (booléen et binaire)	Indexation
Concaténation de chaînes	Cast
Incrément et décrétement	Conditionnel
Déplacement	Concaténation et suppression de délégués
Relationnel	Création d'objet
Assignation	Informations de type
Contrôle d'exception de dépassement	Indirection et adresse

Le langage C# fournit un large éventail d'opérateurs prédéfinis. En voici la liste complète :

Catégorie d'opérateur	Opérateurs
Arithmétique	+, -, *, /, %
Logique (booléen et binaire)	&, , ^, !, ~, &&, , true, false
Concaténation de chaînes	+
Incrément et décrétement	++, --
Déplacement	<<, >>
Relationnel	==, !=, <, >, <=, >=
Assignation	=, +=, -=, *=, /=, %=, &=, =, <<=, >>=
Accès au membre	.
Indexation	[]
Cast	()
Conditionnel	? :
Concaténation et suppression de délégués	+, -
Création d'objet	new
Informations de type	is, sizeof, typeof
Contrôle d'exception de dépassement	checked, unchecked
Indirection et adresse	*, ->, [], &

Vous utilisez les opérateurs pour générer des expressions. Leur fonction est généralement facile à comprendre. Par exemple, l'opérateur d'addition (+) dans l'expression `10 + 5` effectue une addition arithmétique et, dans cet exemple, l'expression retourne la valeur 15.

Il se peut que d'autres opérateurs ne soient pas aussi connus ; certains sont définis sous la forme de mots clés plutôt que de symboles, mais leurs fonctionnalités avec les classes et types de données fournis avec le .NET Framework sont totalement définies.

Opérateurs à plusieurs définitions

Un aspect déroutant des opérateurs est que le même symbole peut posséder plusieurs significations. Le signe + dans l'expression `10 + 5` est clairement l'opérateur d'addition arithmétique. Vous pouvez déterminer la signification selon le contexte dans lequel il est utilisé (+ n'est associé à aucune autre signification).

Cependant, l'exemple suivant utilise l'opérateur + pour concaténer des chaînes :

```
"Adam " + "Barr"
```

C'est l'analyseur qui, une fois le programme compilé, est chargé de déterminer la signification d'un opérateur dans un contexte donné.

◆ Surcharge d'opérateurs

- Introduction à la surcharge d'opérateurs
- Surcharge des opérateurs relationnels
- Surcharge des opérateurs logiques
- Surcharge des opérateurs de conversion
- Surcharge répétée d'opérateurs
- Quiz : Trouver les bogues

De nombreux opérateurs prédéfinis en C# exécutent des fonctions spécifiques sur des classes et d'autres types de données. Cette définition claire élargit la portée de l'expression pour l'utilisateur. Vous pouvez redéfinir certains opérateurs fournis par C# et les utiliser comme opérateurs qui ne fonctionnent qu'avec des classes et des structs que vous déterminez. Dans un sens, cela revient à définir vos propres opérateurs. Ce processus est appelé surcharge des opérateurs.

Tous les opérateurs C# prédéfinis ne peuvent pas être surchargés. Les opérateurs arithmétiques et logiques unaires peuvent être librement surchargés, tout comme les opérateurs arithmétiques binaires. Les opérateurs d'assignation ne peuvent pas être surchargés directement, mais ils sont tous évalués à l'aide des opérateurs arithmétiques, logiques et de déplacement, qui peuvent être à leur tour surchargés.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- surcharger des opérateurs relationnels, logiques et de conversion ;
- surcharger un opérateur à plusieurs reprises.

Introduction à la surcharge d'opérateurs

- **Surcharge d'opérateurs**
 - Définissez vos propres opérateurs en cas de nécessité uniquement
- **Syntaxe des opérateurs**
 - `Operatorop`, où `op` correspond à l'opérateur surchargé
- **Exemple**

```
public static Time operator+(Time t1, Time t2)
{
    int newHours = t1.hours + t2.hours;
    int newMinutes = t1.minutes + t2.minutes;
    return new Time(newHours, newMinutes);
}
```

Même si les opérateurs simplifient les expressions, il est conseillé de n'en définir qu'en cas de nécessité. Les opérateurs ne doivent être surchargés que si la classe ou le **struct** est une donnée (telle qu'un nombre), qui seront utilisés comme tels. L'utilisation d'un opérateur ne doit jamais être ambiguë. Il ne doit exister qu'une seule interprétation possible de sa signification. Par exemple, ne définissez pas d'opérateur d'incrément (`++`) sur une classe **Employee** (`emp1++`), car la sémantique d'une telle opération sur un employé n'est pas claire. Que signifie réellement « incrémenter un employé » ? Serait-il possible de l'utiliser au sein d'une expression plus longue ? Si le terme incrémenter signifie « donner une promotion à un employé », définissez plutôt une méthode **Promote** (`emp1.Promote()` ;).

Syntaxe de la surcharge d'opérateurs

Tous les opérateurs sont des méthodes statiques publiques et leurs noms respectent un modèle particulier. Tous les opérateurs sont appelés `operatorop`, où `op` spécifie exactement l'opérateur qui fait l'objet de la surcharge. Par exemple, `operator+` est la méthode permettant de surcharger l'opérateur d'addition.

Les paramètres de l'opérateur et les types des paramètres qu'il retourne doivent être bien définis. Tous les opérateurs arithmétiques retournent une instance de la classe et manipulent les objets qu'elle contient.

Exemple

Prenons l'exemple du **struct Time** illustré dans le code suivant. Une valeur **Time** est constituée de deux parties : un nombre d'heures et un nombre de minutes. Le code en gras indique comment implémenter l'opérateur d'addition binaire (+) pour additionner deux valeurs **Time**.

```
public struct Time
{
    public Time(int minutes) : this(0, minutes)
    {
    }

    public Time(int hours, int minutes)
    {
        this.hours = hours;
        this.minutes = minutes;
        Normalize( );
    }

    // Arithmétique

    public static Time operator+(Time lhs, Time rhs)
    {
        return new Time(lhs.hours + rhs.hours,
                        lhs.minutes + rhs.minutes
                        );
    }

    public static Time operator-(Time lhs, Time rhs)
    {
        ...
    }

    ...

    // Méthodes d'assistance

    private void Normalize( )
    {
        if (hours < 0 || minutes < 0) {
            throw new ArgumentException("Durée trop courte");
        }
        hours += (minutes / 60);
        minutes %= 60;
    }

    private int TotalMinutes( )
    {
        return hours * 60 + minutes;
    }

    private int hours;
    private int minutes;
}
```


Surcharge des opérateurs relationnels

- Les opérateurs relationnels doivent être associés par paire
 - < et >
 - <= et >=
 - == et !=
- Substituez la méthode **Equals** si vous surchargez == et !=
- Substituez la méthode **GetHashCode** si vous substituez la méthode **Equals**

Vous devez surcharger les opérateurs relationnels ou de comparaison par paires. Chaque opérateur relationnel doit être défini avec son antonyme logique. En d'autres termes, si vous surchargez <, vous devez également surcharger >, et inversement. De même, != doit être surchargé avec ==, et <= avec >=.

Conseil Par souci de cohérence, créez d'abord une méthode **Compare** et définissez tous les opérateurs relationnels à l'aide de celle-ci. L'exemple de code de la page suivante montre comment effectuer cette opération.

Surcharge de la méthode **Equals**

Si vous surchargez == et !=, vous devez également substituer la méthode virtuelle **Equals** dont votre classe hérite de **Object**. Cela garantit la cohérence lors de la comparaison de deux objets de cette classe, que ce soit par == ou par la méthode **Equals**, afin d'éviter toute situation dans laquelle == retournerait la valeur **true** et la méthode **Equals**, la valeur **false**.

Surcharge de la méthode **GetHashCode**

La méthode **GetHashCode** (également héritée de **Object**) est utilisée pour identifier une instance de votre classe si elle est stockée dans une table de hachage. Le hachage de deux instances de la même classe pour lesquelles **Equals** retourne **true** doit également fournir la même valeur entière. Par défaut, cela n'est pas le cas. Par conséquent, si vous substituez la méthode **Equals**, vous devez également substituer la méthode **GetHashCode**.

Exemple

Le code suivant illustre l'implémentation des opérateurs relationnels, de la méthode **Equals** et de la méthode **GetHashCode** pour le **struct Time** :

```
public struct Time
{
    ...

    // Égalité

    public static bool operator==(Time lhs, Time rhs)
    {
        return lhs.Compare(rhs) == 0;
    }

    public static bool operator!=(Time lhs, Time rhs)
    {
        return lhs.Compare(rhs) != 0;
    }

    // Relationnel

    public static bool operator<(Time lhs, Time rhs)
    {
        return lhs.Compare(rhs) < 0;
    }

    public static bool operator>(Time lhs, Time rhs)
    {
        return lhs.Compare(rhs) > 0;
    }

    public static bool operator<=(Time lhs, Time rhs)
    {
        return lhs.Compare(rhs) <= 0;
    }

    public static bool operator>=(Time lhs, Time rhs)
    {
        return lhs.Compare(rhs) >= 0;
    }
}
```

(suite du code à la page suivante)

```
// Méthodes virtuelles héritées (de Object)

public override bool Equals(object obj)
{
    return obj is Time && Compare((Time)obj) == 0;
}

public override int GetHashCode( )
{
    return TotalMinutes( );
}

private int Compare(Time other)
{
    int lhs = TotalMinutes( );
    int rhs = other.TotalMinutes( );

    int result;
    if (lhs < rhs)
        result = -1;
    else if (lhs > rhs)
        result = +1;
    else
        result = 0;

    return result;
}
...
}
```

Surcharge des opérateurs logiques

- **Les opérateurs `&&` et `||` ne peuvent pas être surchargés directement**

- Ils sont évalués en termes d'opérateurs `&`, `|`, **true** et **false**, qui peuvent être surchargés
- **`x && y` est évalué comme `T.false(x) ? x : T.&(x, y)`**
- **`x || y` est évalué comme `T.true(x) ? x : T.|(x, y)`**

Vous ne pouvez pas surcharger directement les opérateurs logiques `&&` et `||`. Cependant, ils sont évalués en termes d'opérateurs `&`, `|`, **true** et **false**, que vous pouvez surcharger.

Si les variables `x` et `y` sont toutes les deux de type **T**, les opérateurs logiques sont évalués de la manière suivante :

- **`x && y` est évalué comme `T.false(x) ? x : T.&(x, y)`**

Cette expression se traduit par « si `x` a la valeur false comme défini par l'opérateur **false** de **T**, le résultat est `x` ; sinon, il s'agit du résultat de l'utilisation de l'opérateur `&` de **T** sur `x` et `y` ».

- **`x || y` est évalué comme `T.true(x) ? x : T.|(x, y)`**

Cette expression signifie que « si `x` a la valeur true comme défini par l'opérateur **true** de **T**, le résultat est `x` ; sinon, il s'agit du résultat de l'utilisation de l'opérateur `|` de **T** sur `x` et `y` ».

Surcharge des opérateurs de conversion

■ Opérateurs de conversion surchargés

```
public static explicit operator Time (float hours)
{ ... }
public static explicit operator float (Time t1)
{ ... }
public static implicit operator string (Time t1)
{ ... }
```

■ Si une classe définit un opérateur de conversion de chaîne

- La classe doit substituer ToString

Vous pouvez définir des opérateurs de conversion implicites et explicites pour vos propres classes, ou encore créer des opérateurs de cast définis par le programmeur qui peuvent être utilisés pour convertir des données d'un type en un autre. Voici quelques exemples d'opérateurs de conversion surchargés :

■ explicit operator Time (int minutes)

Cet opérateur convertit une valeur **int** en une valeur **Time**. Cette conversion est explicite, car toutes les valeurs **int** ne peuvent pas être converties. Un argument négatif entraîne la levée d'une exception.

■ explicit operator Time (float minutes)

Cet opérateur convertit une valeur **float** en une valeur **Time**. À nouveau, il s'agit d'une conversion explicite, car un paramètre négatif entraîne la levée d'une exception.

■ implicit operator int (Time t1)

Cet opérateur convertit une valeur **Time** en une valeur **int**. Il s'agit d'une conversion implicite, car toutes les valeurs **Time** peuvent être converties en toute sécurité en valeurs **int**.

■ explicit operator float (Time t1)

Cet opérateur convertit une valeur **Time** en une valeur **float**. Dans ce cas, l'opérateur est explicite, car toutes les valeurs **Time** peuvent être converties en valeurs **float**. Il se peut que la représentation à virgule flottante de certaines valeurs ne soit pas exacte. (Vous prenez toujours ce risque avec les calculs impliquant des valeurs à virgule flottante.)

■ implicit operator string (Time t1)

Cet opérateur convertit une valeur **Time** en une valeur **string**. Il s'agit également d'une conversion implicite, car il n'existe aucun danger de perte d'information lors de la conversion.

Substitution de la méthode ToString

Par souci de cohérence, les principes de conception recommandent que toute classe possédant un opérateur de conversion de chaîne substitue la méthode **ToString**, qui doit exécuter la même fonction. De nombreuses classes et méthodes au sein de l'espace de noms **System** (**Console.WriteLine**, par exemple) utilisent la méthode **ToString** pour créer une version imprimable d'un objet.

Exemple

Le code suivant montre comment implémenter les opérateurs de conversion. Il montre également un moyen d'implémenter la méthode **ToString**. Remarquez la manière dont le **struct Time** substitue **ToString**, qui est hérité de **Object**.

```
public struct Time
{
    ...

    // Opérateurs de conversion
    public static explicit operator Time (int minutes)
    {
        return new Time(0, minutes);
    }

    public static explicit operator Time (float minutes)
    {
        return new Time(0, (int)minutes);
    }

    public static implicit operator int (Time t1)
    {
        return t1.TotalMinutes( );
    }

    public static explicit operator float (Time t1)
    {
        return t1.TotalMinutes( );
    }

    public static implicit operator string (Time t1)
    {
        return t1.ToString( );
    }

    // Méthodes virtuelles héritées (de Object)

    public override string ToString( )
    {
        return String.Format("{0}:{1:00}", hours, minutes);
    }
    ...
}
```

Conseil Si un opérateur de conversion peut lever une exception ou retourner un résultat partiel, rendez-le explicite. Si une conversion est garantie comme fonctionnant sans perte de données, vous pouvez la rendre implicite.

Surcharge répétée d'opérateurs

- **Le même opérateur peut être surchargé à plusieurs reprises**

```
public static Time operator+(Time t1, int hours)
{...}

public static Time operator+(Time t1, float hours)
{...}

public static Time operator-(Time t1, int hours)
{...}

public static Time operator-(Time t1, float hours)
{...}
```

Vous pouvez surcharger le même opérateur à plusieurs reprises pour fournir d'autres implémentations qui acceptent différents types comme paramètres. Au moment de la compilation, le système établit la méthode à appeler selon les types de paramètres utilisés pour appeler l'opérateur.

Exemple

Le code suivant montre d'autres exemples de l'implémentation des opérateurs + et – pour le **struct Time**. Les deux exemples ajoutent ou soustraient un nombre d'heures spécifié à la valeur **Time** fournie :

```
public struct Time
{
    ...
    public static Time operator+(Time t1, int hours)
    {
        return t1 + new Time(hours, 0);
    }

    public static Time operator+(Time t1, float hours)
    {
        return t1 + new Time((int)hours, 0);
    }

    public static Time operator-(Time t1, int hours)
    {
        return t1 - new Time(hours, 0);
    }

    public static Time operator-(Time t1, float hours)
    {
        return t1 - new Time((int)hours, 0);
    }
    ...
}
```


Quiz : Trouver les bogues

```
public bool operator != (Time t1, Time t2)
{ ... }
```

1

```
public static operator float(Time t1) { ... }
```

2

```
public static Time operator += (Time t1, Time t2)
{ ... }
```

3

```
public static bool Equals(Object obj) { ... }
```

4

```
public static int operator implicit(Time t1)
{ ... }
```

5

Vous pouvez travailler avec un partenaire pour trouver les bogues du code reproduit sur la diapositive. Les réponses se trouvent à la page suivante.

Réponses

1. Les opérateurs doivent être statiques, car ils appartiennent à la classe, et non à un objet. La définition de l'opérateur `!=` doit être la suivante :

```
public static bool operator != (Time t1, Time t2) { ... }
```
2. Le « type » est absent. Les opérateurs de conversion doivent être implicites ou explicites. Le code doit ressembler à ce qui suit :

```
public static implicit operator float (Time t1) { ... }
```
3. Vous ne pouvez pas surcharger l'opérateur `+=`. Cependant, `+=` est évalué à l'aide de l'opérateur `+`, que vous pouvez surcharger.
4. La méthode **Equals** doit être une méthode d'instance et non une méthode de classe. Cependant, si vous supprimez le mot clé **static**, cette méthode masque la méthode virtuelle héritée de **Object** et n'est pas appelée de la manière escomptée. Par conséquent, le code doit plutôt utiliser **override**, comme suit :

```
public override bool Equals(Object obj) { ... }
```
5. Les mots clés **int** et **implicit** ont été transposés. Le nom de l'opérateur doit être **int**, et son type doit être implicite, comme suit :

```
public static implicit operator int(Time t1) { ... }
```

Remarque Toutes les situations énumérées ci-dessus entraînent des erreurs de compilation.

Atelier 12.1 : Définition d'opérateurs



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer des opérateurs pour l'addition, la soustraction, les tests d'égalité, la multiplication, la division et le cast ;
- substituer les méthodes **Equals**, **ToString** et **GetHashCode**.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les éléments suivants :

- utilisation de l'héritage en C# ;
- définition de constructeurs et de destructeurs ;
- compilation et utilisation d'assemblies ;
- opérateurs C# de base.

Durée approximative de cet atelier : 30 minutes

Exercice 1

Définition d'opérateurs pour la classe **BankAccount**

Dans les ateliers précédents, vous avez créé des classes pour un système bancaire. La classe **BankAccount** contient des informations sur les comptes bancaires des clients, notamment leur numéro de compte et leur solde. Vous avez également créé une classe **Bank** qui joue le rôle de fabrique pour la création et la gestion des objets **BankAccount**.

Dans cet exercice, vous allez définir les opérateurs `==` et `!=` dans la classe **BankAccount**. L'implémentation par défaut de ces opérateurs, qui est héritée de **Object**, effectue un test pour vérifier si les références sont identiques. Vous devez les redéfinir pour examiner et comparer les informations de deux comptes.

Vous devez ensuite substituer les méthodes **Equals** et **ToString**. La méthode **Equals** est utilisée par de nombreuses parties du runtime et doit se comporter de la même manière que les opérateurs d'égalité. De nombreuses classes du .NET Framework utilisent la méthode **ToString** lorsqu'elles ont besoin qu'un objet soit représenté sous forme de chaîne. Utilisez le fichier de démarrage fourni pour cet atelier.

► Pour définir les opérateurs `==` et `!=`

1. Ouvrez le projet `Bank.sln` situé dans le dossier *dossier d'installation\Labs\Lab12\Starter\Bank*.
2. Ajoutez la méthode suivante à la classe **BankAccount** :

```
public static bool operator == (BankAccount acc1,
BankAccount acc2)
{
    ...
}
```
3. Dans le corps de **operator ==**, ajoutez des instructions pour comparer les deux objets **BankAccount**. Si le numéro de compte, le type et le solde des deux comptes sont identiques, **true** est retourné ; sinon, c'est **false** qui est retourné.
4. Compilez le projet. Une erreur s'affiche.
(Pourquoi rencontrez-vous une erreur lorsque vous compilez le projet ?)
5. Ajoutez la méthode suivante à la classe **BankAccount** :

```
public static bool operator != (BankAccount acc1,
BankAccount acc2)
{
    ...
}
```
6. Ajoutez des instructions dans le corps de **operator !=** pour comparer le contenu des deux objets **BankAccount**. Si le numéro de compte, le type et le solde des deux comptes sont identiques, **false** est retourné ; sinon, c'est **true** qui est retourné. Vous pouvez aussi procéder en appelant **operator ==** et en inversant le résultat.

7. Enregistrez et compilez le projet. Le projet doit maintenant être compilé correctement. L'erreur précédente a été provoquée par l'absence d'une méthode **operator ==** correspondante. (Si vous définissez **operator ==**, vous devez également définir **operator !=**, et inversement.)

Le code complet des deux opérateurs est le suivant :

```
public class BankAccount
{
    ...
    public static bool operator == (BankAccount acc1,
    ↪BankAccount acc2)

    {
        if ((acc1.accNo == acc2.accNo) &&
            (acc1.accType == acc2.accType) &&
            (acc1.accBal == acc2.accBal)) {
            return true;
        } else {
            return false;
        }
    }

    public static bool operator != (BankAccount acc1,
    ↪BankAccount acc2)

    {
        return !(acc1 == acc2);
    }
    ...
}
```

► Pour tester les opérateurs

1. Ouvrez le projet TestHarness.sln situé dans le dossier *dossier d'installation\Labs\Lab12\Starter\TestHarness*.
2. Créez une référence au composant **Bank** que vous avez créé dans les ateliers précédents. Pour ce faire :
 - a. Développez le projet TestHarness dans l'Explorateur de solutions.
 - b. Cliquez avec le bouton droit sur **Références**, puis cliquez sur **Ajouter une référence**.
 - c. Cliquez sur **Parcourir** et accédez au *dossier d'installation\Labs\Lab12\Starter\Bank\bin\debug*.
 - d. Cliquez sur **Bank.dll**, puis sur **Ouvrir**.
 - e. Cliquez sur **OK**.
3. Créez deux objets **BankAccount** dans la méthode **Main** de la classe **CreateAccount**. Pour ce faire :
 - a. Utilisez **Bank.CreateAccount()** et instanciez les objets **BankAccount** avec le même solde et le même type de compte.
 - b. Stockez les numéros de compte générés dans deux variables de type long appelées *accNo1* et *accNo2*.

4. Créez deux variables **BankAccount** appelées *acc1* et *acc2*. Remplissez-les avec les deux comptes créés à l'étape précédente en appelant **Bank.GetAccount()**.
5. Comparez *acc1* et *acc2* à l'aide de l'opérateur **==**. Ce test doit retourner la valeur **false**, car les deux comptes possèdent un numéro de compte différent.
6. Comparez *acc1* et *acc2* à l'aide de l'opérateur **!=**. Ce test doit retourner la valeur **true**.
7. Créez une troisième variable **BankAccount** appelée *acc3*. Remplissez-la avec le compte que vous avez utilisé pour remplir *acc1* en appelant **Bank.GetAccount()**, avec *accNo1* comme paramètre.
8. Comparez *acc1* et *acc3* à l'aide de l'opérateur **==**. Ce test doit retourner la valeur **true**, car les deux comptes possèdent les mêmes données.
9. Comparez *acc1* et *acc3* à l'aide de l'opérateur **!=**. Ce test doit retourner la valeur **false**.

En cas de problème, vous disposez d'une fonction utilitaire appelée **Write** que vous pouvez employer pour afficher le contenu d'une variable **BankAccount** passée comme paramètre.

Votre code complet pour le test de validation doit être le suivant :

```
class CreateAccount
{
    static void Main( )
    {

        long accNo1 = Bank.CreateAccount(AccountType.Courant,
                                           100);
        long accNo2 = Bank.CreateAccount(AccountType.Courant,
                                           100);

        BankAccount acc1 = Bank.GetAccount(accNo1);
        BankAccount acc2 = Bank.GetAccount(accNo2);

        if (acc1 == acc2) {
            Console.WriteLine(
                ↪ "Les deux comptes sont identiques. Ils ne devraient
                ↪ pas l'être !");
        } else {
            Console.WriteLine(
                ↪ "Les comptes sont différents. Parfait !");
        }

        if (acc1 != acc2) {
            Console.WriteLine(
                ↪ "Les comptes sont différents. Parfait !");
        } else {
            Console.WriteLine(
                ↪ "Les deux comptes sont identiques. Ils ne devraient
                ↪ pas l'être !");
        }
    }
}
```

(suite du code à la page suivante)

```

        BankAccount acc3 = Bank.GetAccount(accNo1);
        if (acc1 == acc3) {
            Console.WriteLine(
↪      "Les comptes sont identiques. Parfait !");
        } else {
            Console.WriteLine(
↪      "Les comptes sont différents. Ils ne devraient pas
↪      l'être !");
        }

        if (acc1 != acc3) {
            Console.WriteLine(
↪      "Les comptes sont différents. Ils ne devraient pas
↪      l'être !");
        } else {
            Console.WriteLine(
↪      "Les comptes sont identiques. Parfait !");
        }
    }
}

```

10. Compilez et exécutez le test de validation.

► Pour substituer les méthodes **Equals**, **ToString** et **GetHashCode**

1. Ouvrez le projet Bank.sln situé dans le dossier *dossier d'installation\Labs\Lab12\Starter\Bank*.
2. Ajoutez la méthode **Equals** à la classe **BankAccount** :

```

public override bool Equals(object acc1)
{
    ...
}

```

La méthode **Equals** doit fonctionner comme l'opérateur **==**, excepté qu'elle est une méthode d'instance et non de classe. Utilisez l'opérateur **==** pour comparer **this** à *acc1*.

3. Ajoutez la méthode **ToString** comme suit :

```

public override string ToString( )
{
    ...
}

```

Le corps de la méthode **ToString** doit retourner une représentation de l'instance sous forme de chaîne.

4. Ajoutez la méthode **GetHashCode** comme suit :

```
public override int GetHashCode( )
{
    ...
}
```

La méthode **GetHashCode** doit retourner une valeur unique pour chaque compte différent, mais des références différentes au même compte doivent retourner la même valeur. La solution la plus simple consiste à retourner le numéro de compte. (Vous devez d'abord le caster en une valeur de type **int**.)

5. Le code complet de **Equals**, **ToString** et **GetHashCode** est le suivant :

```
public override bool Equals(Object acc1)
{
    return this == (BankAccount)acc1;
}

public override string ToString( )
{
    string retVal = "Numéro : " + this.accNo + "\tType : ";
    retVal += (this.accType == AccountType.Courant) ?
    ↪ "Courant" : "Epargne";
    retVal += "\tSolde : " + this.accBal;

    return retVal;
}

public override int GetHashCode( )
{
    return (int)this.accNo;
}
```

6. Enregistrez et compilez le projet. Corrigez les erreurs, le cas échéant.

► **Pour tester les méthodes Equals et ToString**

1. Ouvrez le projet TestHarness.sln situé dans le dossier *dossier d'installation*\Labs\Lab12\Starter\TestHarness.
2. Dans la méthode **Main** de la classe **CreateAccount**, remplacez l'utilisation de `==` et `!=` par **Equals**, comme suit :

```
if (acc1.Equals(acc2)) {  
    ...  
}  
  
if (!acc1.Equals(acc2)) {  
    ...  
}
```

3. Après les instructions **if**, ajoutez trois instructions **WriteLine** qui affichent le contenu de *acc1*, *acc2* et *acc3*, comme l'illustre le code suivant. La méthode **WriteLine** utilise **ToString** pour formater ses arguments comme chaînes.

```
Console.WriteLine("acc1 - {0}", acc1);  
Console.WriteLine("acc2 - {0}", acc2);  
Console.WriteLine("acc3 - {0}", acc3);
```

4. Appelez la méthode **Dispose** pour chaque objet de compte.
5. Compilez et exécutez le test de validation. Vérifiez les résultats.

Exercice 2

Gestion des nombres rationnels

Dans cet exercice, vous allez créer une classe totalement nouvelle pour la gestion des nombres rationnels. Quittons brièvement l'univers bancaire.

Un nombre rationnel est un nombre qui peut être écrit sous la forme d'un rapport de deux entiers (par exemple, $\frac{1}{2}$, $\frac{3}{4}$ et -17). Vous allez créer une classe **Rational**, qui sera constituée d'une paire de variables d'instances entières privées (appelées *dividend* et *divisor*) ainsi que d'opérateurs permettant d'effectuer des calculs et des comparaisons sur celles-ci. Les opérateurs et méthodes ci-dessous seront définis :

- **Rational(int dividend)**
Il s'agit d'un constructeur qui affecte la valeur fournie à *dividend* et la valeur 1 à *divisor*.
- **Rational(int dividend, int divisor)**
Il s'agit d'un constructeur qui définit *dividend* et *divisor*.
- **== et !=**
Ces opérateurs effectuent des comparaisons basées sur la valeur numérique calculée des deux opérandes (par exemple, `Rational(6, 8) == Rational(3, 4)`). Vous devez substituer les méthodes **Equals()** pour effectuer la même comparaison.
- **<, >, <=, >=**
Ces opérateurs effectuent les comparaisons relationnelles appropriées entre deux nombres rationnels (par exemple, `Rational(6, 8) > Rational(1, 2)`).
- **+ et - binaires**
Ces opérateurs ajoutent un nombre rationnel à un autre ou le soustraient.
- **++ et --**
Ces opérateurs incrémentent et décrémentent le nombre rationnel.

► Pour créer les constructeurs et la méthode ToString

1. Ouvrez le projet `Rational.sln` situé dans le dossier *dossier d'installation\Labs\Lab12\Starter\Rational*.
2. La classe **Rational** contient deux types de variables d'instances privées appelées *dividend* et *divisor*. Elles sont initialisées à 0 et 1, respectivement. Ajoutez un constructeur qui prend un seul nombre entier et l'utilise pour définir *dividend*, en laissant la valeur 1 à *divisor*.
3. Ajoutez un autre constructeur qui prend deux nombres entiers. Le premier est assigné à *dividend*, et le second à *divisor*. Vérifiez que *divisor* n'a pas la valeur zéro. Levez une exception si cela se produit et déclenchez **ArgumentOutOfRangeException**.

4. Créez un troisième constructeur qui prend *Rational* comme paramètre et copie les valeurs qu'il contient.

Remarque Les développeurs C++ reconnaîtront le troisième constructeur comme étant un constructeur de copie. Vous l'utiliserez ultérieurement dans cet atelier.

Le code complet des trois constructeurs est le suivant :

```
public Rational(int dividend)
{
    this.dividend = dividend;
    this.divisor = 1;
}

public Rational(int dividend, int divisor)
{
    if (divisor == 0) {
        throw new ArgumentOutOfRangeException(
            "Le diviseur ne peut pas être zéro");
    } else {
        this.dividend = dividend;
        this.divisor = divisor;
    }
}

public Rational(Rational r1)
{
    this.dividend = r1.dividend;
    this.divisor = r1.divisor;
}
```

5. Substituez la méthode **ToString** qui retourne une version **string** de *Rational*, comme suit :

```
public override string ToString( )
{
    return String.Format("{0}/{1}", dividend, divisor);
}
```

6. Compilez le projet et corrigez les erreurs, le cas échéant.

► Pour définir les opérateurs relationnels

1. Dans la classe **Rational**, créez l'opérateur `==` comme suit :

```
public static bool operator == (Rational r1, Rational r2)
{
    ...
}
```

2. L'opérateur `==` effectue les opérations suivantes :

- a. Il établit la valeur décimale de *r1* à l'aide de la formule suivante :

```
decimalValue1 = r1.dividend / r1.divisor
```

- b. Il établit la valeur décimale de *r2* à l'aide d'une formule similaire.

- c. Il compare les deux valeurs décimales et retourne **true** ou **false**, selon le cas. Le code complet est le suivant :

```
public static bool operator == (Rational r1, Rational
r2)
{
    decimal decimalValue1 =
        (decimal)r1.dividend / r1.divisor;
    decimal decimalValue2 =
        (decimal)r2.dividend / r2.divisor;
    return decimalValue1 == decimalValue2;
}
```

Remarque Ce code peut être réduit à :

```
public static bool operator == (Rational r1, Rational r2)
{
    return (r1.dividend * r2.divisor) == (r2.dividend *
r1.divisor);
}
```

Pourquoi les casts en decimal sont-ils nécessaires lors de l'exécution de la division ?

3. Créez et définissez l'opérateur `!=` à l'aide de l'opérateur `==`, comme suit :

```
public static bool operator != (Rational r1, Rational r2)
{
    return !(r1 == r2);
}
```

4. Substituez la méthode **Equals**. Utilisez l'opérateur `==`, comme suit :

```
public override bool Equals(Object r1)
{
    return (this == (Rational)r1);
}
```

5. Définissez l'opérateur <. Optez pour une stratégie similaire à celle utilisée pour l'opérateur ==, comme suit :

```
public static bool operator < (Rational r1, Rational r2)
{
    return (r1.dividend * r2.divisor) < (r2.dividend *
↪r1.divisor);
}
```

6. Créez l'opérateur > à l'aide de == et <, comme l'illustre le code suivant. Veillez à bien comprendre la logique booléenne utilisée par l'expression dans l'instruction de retour.

```
public static bool operator > (Rational r1, Rational r2)
{
    return !((r1 < r2) || (r1 == r2));
}
```

7. Définissez les opérateurs <= et >= en termes de > et <, comme l'illustre le code suivant :

```
public static bool operator <= (Rational r1, Rational r2)
{
    return !(r1 > r2);
}

public static bool operator >= (Rational r1, Rational r2)
{
    return !(r1 < r2);
}
```

8. Compilez le projet et corrigez les erreurs, le cas échéant.

► **Pour tester les constructeurs, la méthode ToString et les opérateurs relationnels**

1. Dans la méthode **Main** de la classe **TestRational** du projet Rational, créez deux variables **Rational**, *r1* et *r2*, et instanciez-les avec les paires de valeurs (1,2) et (1,3), respectivement.
2. Affichez-les à l'aide de **WriteLine** pour tester la méthode **ToString**.
3. Effectuez les comparaisons suivantes et affichez un message indiquant les résultats :
 - a. *r1* > *r2* ?
 - b. *r1* <= *r2* ?
 - c. *r1* != *r2* ?
4. Compilez et exécutez le programme. Vérifiez les résultats.
5. Modifiez la variable *r2* et instanciez-la avec la paire de valeurs (2,4).
6. Compilez et exécutez à nouveau le programme. Vérifiez les résultats.

► **Pour créer les opérateurs d'addition binaires**

1. Dans la classe **Rational**, créez l'opérateur + binaire. Créez deux versions pour :
 - a. Ajouter deux nombres de type **Rational**.

Conseil Pour additionner deux nombres rationnels, vous devez déterminer un dénominateur (diviseur) commun. À moins que les deux diviseurs ne soient identiques (s'ils le sont, vous pouvez ignorer cette étape et la suivante), effectuez cette opération en multipliant les diviseurs l'un par l'autre. Par exemple, supposons que vous souhaitez ajouter $1/4$ à $2/3$. Le diviseur commun est 12 ($4 * 3$). La prochaine étape consiste à multiplier le dividende de chaque nombre par le diviseur de l'autre. Ainsi, $1/4$ devient $(1 * 3)/12$, ou $3/12$, et $2/3$ devient $(4 * 2)/12$, ou $8/12$. Pour terminer, vous devez additionner les deux dividendes et utiliser le diviseur commun. $3/12 + 8/12 = 11/12$, et par conséquent $1/4 + 2/3 = 11/12$. Si vous utilisez cet algorithme, vous devez effectuer des copies des paramètres passés (à l'aide du constructeur de copie défini précédemment) à l'opérateur +. Si vous modifiez les paramètres formels, vous pouvez constater que les paramètres actuels sont également modifiés, en raison du mode de passage des types référence.

- b. Ajouter un nombre rationnel et un nombre entier.

Conseil Pour ajouter un nombre entier à un nombre rationnel, convertissez le nombre entier en un nombre rationnel possédant le même diviseur. Par exemple, pour additionner 2 et $3/8$, convertissez 2 en $16/8$, puis effectuez l'addition.

Les deux versions doivent retourner un nombre de type **Rational**.
(Ne vous souciez pas de produire un résultat normalisé.)

2. Créez l'opérateur – binaire. Créez deux versions pour :

- a. soustraire un nombre rationnel d'un autre ;
- b. soustraire un nombre entier d'un nombre rationnel.

Les deux versions doivent retourner un nombre de type **Rational** (non normalisé). Le code complet des opérateurs + et – est le suivant :

```
public static Rational operator + (Rational r1, Rational
r2)
{
    // Effectue des copies opérationnelles de r1 et r2
    Rational tempR1 = new Rational(r1);
    Rational tempR2 = new Rational(r2);

    // Détermine un dénominateur commun.
    // Par exemple, pour additionner 1/4 et 2/3, les
    // convertit en 3/12 et 8/12
    int commonDivisor;
    if (tempR1.divisor != tempR2.divisor) {
        commonDivisor = tempR1.divisor * tempR2.divisor;

        // Multiplie les dividendes de chaque rationnel
        tempR1.dividend *= tempR2.divisor;
        tempR2.dividend *= tempR1.divisor;
    } else {
        commonDivisor = tempR1.divisor;
    }

    // Crée un nouveau Rational.
    // Par exemple, 1/4 + 2/3 = 3/12 + 8/12 = 11/12.

    Rational result = new Rational(tempR1.dividend +
        tempR2.dividend, commonDivisor);
    return result;
}

public static Rational operator + (Rational r1, int i1)
{
    // Convertit i1 en un nombre de type Rational
    Rational r2 = new Rational(i1 * r1.divisor,
        r1.divisor);

    // Additionne des nombres rationnels
    return r1 + r2;
}
```

(suite du code à la page suivante)

```
public static Rational operator - (Rational r1, Rational
r2)
{
    // Effectue des copies opérationnelles de r1 et r2
    Rational tempR1 = new Rational(r1);
    Rational tempR2 = new Rational(r2);

    // Détermine un dénominateur commun.
    // Par exemple, pour soustraire 2/3 de 1/4,
    // les convertit en 8/12 et 3/12.
    int commonDivisor;
    if (tempR1.divisor != tempR2.divisor) {
        commonDivisor = tempR1.divisor * tempR2.divisor;

        // Multiplie les dividendes de chaque rationnel
        tempR1.dividend *= tempR2.divisor;
        tempR2.dividend *= tempR1.divisor;
    } else {
        commonDivisor = tempR1.divisor;
    }

    // Crée un nouveau Rational.
    // Par exemple,  $2/3 - 1/4 = 8/12 - 3/12 = 5/12$ .

    Rational result = new Rational(tempR1.dividend -
                                   tempR2.dividend, commonDivisor);
    return result;
}

public static Rational operator - (Rational r1, int i1)
{
    // Convertit i1 en un Rational
    Rational r2 = new Rational(i1 * r1.divisor, r1.divisor);

    // Soustrait un nombre rationnel
    return r1 - r2;
}
```


► Pour définir les opérateurs d'incrément et de décréement

1. Dans la classe **Rational**, créez l'opérateur ++ unaire.

Conseil Utilisez l'opérateur + que vous avez défini précédemment pour ajouter 1 au paramètre passé à l'opérateur ++.

2. Dans la classe **Rational**, créez l'opérateur -- unaire. Le code complet des deux opérateurs est le suivant :

```
public static Rational operator ++ (Rational r1)
{
    return r1 + 1;
}

public static Rational operator -- (Rational r1)
{
    return r1 - 1;
}
```

► Pour tester les opérateurs d'addition

1. Dans la méthode **Main** de la classe **TestRational**, ajoutez les instructions pour :
 - a. Ajouter *r2* à *r1* et afficher le résultat.
 - b. Ajouter 5 à *r2* (utilisez +=) et afficher le résultat.
 - c. Soustraire *r1* de *r2* (utilisez -=) et afficher le résultat.
 - d. Soustraire 2 de *r2* et afficher le résultat.
 - e. Incrémenter *r1* et afficher le résultat.
 - f. Décrémenter *r2* et afficher le résultat.
2. Compilez et exécutez le programme. Vérifiez les résultats.

S'il vous reste du temps

Création d'opérateurs supplémentaires sur nombres rationnels

Dans cet exercice, vous allez créer les opérateurs supplémentaires suivants pour la classe **Rational** :

- Casts explicites et implicites

Il s'agit de conversions entre les types **Rational**, **float** et **int**.

- *****, **/**, **%**

Ces opérateurs de multiplication binaires sont destinés à la multiplication, la division et l'extraction du reste après la division entière de deux nombres rationnels.

► Pour définir les opérateurs de cast

1. Définissez un opérateur de cast explicite pour la conversion d'un nombre rationnel en nombre à virgule flottante, comme suit :

```
public static explicit operator float (Rational r1)
{
    ...
}
```

2. Dans le corps de l'opérateur de cast **float**, retournez le résultat de la division de *dividend* par *divisor*. Vérifiez qu'une division à virgule flottante est effectuée.
3. Créez un opérateur de cast explicite pour la conversion d'un nombre rationnel en nombre entier, comme suit :

```
public static explicit operator int (Rational r1)
{
    ...
}
```

Remarque Cet opérateur est explicite, car une perte d'informations est susceptible de se produire.

4. Dans le corps de l'opérateur de cast **int**, divisez *dividend* par *divisor*. Vérifiez qu'une division à virgule flottante est effectuée. Tronquez le résultat en une valeur de type **int** et retournez-le.
5. Créez un opérateur de cast implicite pour la conversion d'un nombre entier en nombre rationnel, comme suit :

```
public static implicit operator Rational (int i1)
{
    ...
}
```

Remarque Il est conseillé de rendre cet opérateur implicite.

6. Dans le corps de l'opérateur de cast **Rational**, créez une nouvelle valeur **Rational** en affectant la valeur *i1* à *dividend* et la valeur 1 à *divisor*. Retournez cette valeur **Rational**. Le code complet des trois opérateurs de cast est le suivant :

```
public static implicit operator float (Rational r1)
{
    float temp;
    temp = (float)r1.dividend / r1.divisor;
    return (int)temp;
}

public static explicit operator int (Rational r1)
{
    float temp;
    temp = (float)r1.dividend / r1.divisor;
    return (int) temp;
}

public static implicit operator Rational (int i1)
{
    Rational temp = new Rational(i1, 1);
    return temp;
}
```

7. Ajoutez des instructions au test de validation pour tester ces opérateurs.

► Pour définir les opérateurs de multiplication

1. Définissez l'opérateur de multiplication (*) pour multiplier deux nombres rationnels, comme suit :

```
public static Rational operator *(Rational r1, Rational r2)
{
    ...
}
```

Conseil Pour multiplier deux nombres rationnels, vous devez multiplier le dividende et le diviseur des deux nombres rationnels l'un par l'autre.

2. Définissez l'opérateur de division (/) pour diviser un nombre rationnel par un autre, comme suit :

```
public static Rational operator / (Rational r1, Rational
r2)
{
    ...
}
```

Conseil Pour diviser **Rational** *r1* par **Rational** *r2*, multipliez *r1* par l'inverse de *r2*. En d'autres termes, permutuez la valeur *dividend* et la valeur *divisor* de *r2*, puis effectuez la multiplication. ($1/3 / 2/5$ est identique à $1/3 * 5/2$.)

3. Définissez l'opérateur modulo (%). (Le modulo est le reste obtenu après une division.) Il retourne le reste obtenu après la division par un nombre entier :

```
public static Rational operator % (Rational r1, int i1)
{
    ...
}
```

Conseil Convertissez *r1* en une valeur de type **int** appelée *temp*, puis déterminez la différence entre *r1* et *temp*, en stockant le résultat dans une valeur **Rational** appelée *diff*. Exécutez l'opération *temp % i1* et stockez le résultat dans une valeur de type **int** appelée *remainder*. Additionnez *diff* et *remainder*.

Le code complet des opérateurs est le suivant :

```
public static Rational operator * (Rational r1, Rational r2)
{
    int dividend = r1.dividend * r2.dividend;
    int divisor = r1.divisor * r2.divisor;
    Rational temp = new Rational(dividend, divisor);
    return temp;
}

public static Rational operator / (Rational r1, Rational r2)
{
    // Crée l'inverse de r2, puis effectue la multiplication
    Rational temp = new Rational(r2.divisor, r2.dividend);
    return r1 * temp;
}

public static Rational operator % (Rational r1, int i1)
{
    // Convertit r1 en int
    int temp = (int)r1;

    // Calcule la différence d'arrondi entre temp et r1
    Rational diff = r1 - temp;

    // Effectue le modulo (%) de temp par i1
    int remainder = temp % i1;

    // Additionne remainder et diff pour obtenir
    // le résultat complet
    diff += remainder;
    return diff;
}
```

4. Ajoutez des instructions au test de validation pour tester ces opérateurs.

◆ Création et utilisation de délégués

- Scénario : Centrale électrique
- Analyse du problème
- Création de délégués
- Utilisation de délégués

Les délégués vous permettent d'écrire du code capable de modifier de manière dynamique les méthodes qu'il appelle. Il s'agit d'une fonctionnalité souple permettant à une méthode de varier indépendamment du code qui l'appelle.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- analyser un scénario pour lequel les délégués s'avèrent utiles ;
- définir et utiliser des délégués.

Scénario : Centrale électrique

■ Le problème

- Comment répondre aux variations de température dans une centrale électrique
- Plus précisément, lorsque la température du noyau du réacteur s'élève au-dessus d'un certain niveau, des pompes de refroidissement doivent être alertées et actionnées

■ Solutions possibles

- Toutes les pompes de refroidissement doivent-elles contrôler la température du noyau ?
- Un composant qui contrôle le noyau doit-il actionner les pompes appropriées lorsque la température varie ?

Pour comprendre comment employer les délégués, prenons l'exemple d'une centrale électrique pour laquelle l'utilisation d'un délégué est une bonne solution.

Le problème

Dans une centrale électrique, la température du réacteur nucléaire doit être maintenue en dessous d'un niveau critique. Des sondes situées à l'intérieur du noyau la contrôlent en permanence. Si elle augmente de manière significative, différentes pompes doivent être activées pour augmenter le flux de liquide de refroidissement à travers le noyau. Le logiciel qui contrôle le fonctionnement du réacteur nucléaire doit activer les pompes appropriées au moment adéquat.

Solutions possibles

Le logiciel de contrôle peut être conçu de différentes manières permettant de répondre à ces critères. Voici deux solutions possibles :

- Le logiciel pilotant les pompes de refroidissement peut mesurer constamment la température du noyau et augmenter le flux de liquide de refroidissement si nécessaire.
- Le composant qui contrôle la température du noyau peut activer les pompes de refroidissement appropriées à chaque variation de température.

Ces deux techniques présentent des inconvénients. Dans la première, il est nécessaire de déterminer la fréquence de mesure de la température. Des mesures trop fréquentes peuvent affecter le fonctionnement des pompes, car le logiciel doit actionner ces dernières ainsi que contrôler la température du noyau. Des mesures trop espacées dans le temps peuvent empêcher de détecter à temps une augmentation très rapide de la température.

Dans la deuxième technique, il se peut que des dizaines de pompes et de contrôleurs doivent être alertés de chaque variation de température. La programmation requise pour atteindre cet objectif peut être complexe et difficile à gérer, particulièrement s'il existe dans le système différents types de pompes qui doivent être alertées de différentes manières.

Analyse du problème

■ Problèmes existants

- Il peut y avoir plusieurs types de pompes, fournis par différents fabricants
- Chaque pompe doit posséder sa propre méthode d'activation

■ Préoccupations futures

- Pour ajouter une nouvelle pompe, l'intégralité du code devra être modifiée
- Chaque ajout supplémentaire entraînera des coûts élevés

■ Une solution

- Utilisez des délégués dans votre code

Pour commencer à résoudre le problème, tenez compte de la dynamique d'implémentation d'une solution dans le scénario de la centrale électrique.

Problèmes existants

Le principal problème réside dans le fait qu'il peut exister plusieurs types de pompes de marques différentes, possédant chacune son propre logiciel de contrôle. Pour chaque type de pompe, le composant contrôlant la température du noyau devra reconnaître la méthode à appeler pour activer la pompe.

Pour cet exemple, supposons qu'il existe deux types de pompes : électrique et pneumatique. Chaque type de pompe possède son propre pilote logiciel contenant une méthode d'activation de la pompe, à savoir :

```
public class ElectricPumpDriver
{
    ...
    public void StartElectricPumpRunning( )
    {
        ...
    }
}

public class PneumaticPumpDriver
{
    ...
    public void SwitchOn( )
    {
        ...
    }
}
```


Le composant contrôlant la température du noyau actionne la pompe. Le code suivant représente la partie principale de ce composant, la classe **CoreTempMonitor**. Il crée une série de pompes et les stocke dans une **ArrayList**, classe de collection qui implémente un tableau de longueur variable. La méthode **SwitchOnAllPumps** parcourt l'**ArrayList**, détermine le type de pompe et appelle la méthode appropriée pour activer la pompe :

```
public class CoreTempMonitor
{
    public void Add(object pump)
    {
        pumps.Add(pump);
    }

    public void SwitchOnAllPumps()
    {
        foreach (object pump in pumps) {
            if (pump is ElectricPumpDriver) {
                ((ElectricPumpDriver)pump).StartElectricPumpRunning();
            }
            if (pump is PneumaticPumpDriver) {
                ((PneumaticPumpDriver)pump).SwitchOn();
            }
            ...
        }
    }
    ...
    private ArrayList pumps = new ArrayList();
}

public class ExampleOfUse
{
    public static void Main( )
    {
        CoreTempMonitor ctm = new CoreTempMonitor();

        ElectricPumpDriver ed1 = new ElectricPumpDriver();
        ctm.Add(ed1);

        PneumaticPumpDriver pd1 = new PneumaticPumpDriver();
        ctm.Add(pd1);

        ctm.SwitchOnAllPumps();
    }
}
```

Préoccupations futures

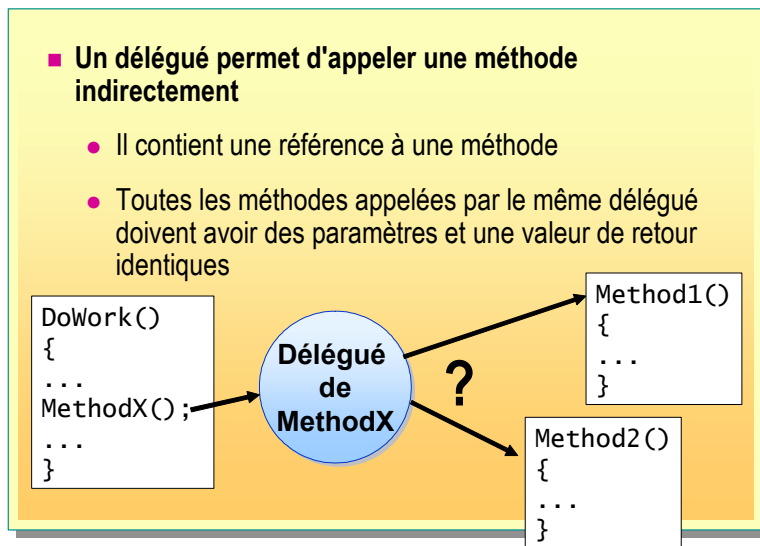
L'utilisation de la structure ci-dessus présente un sérieux inconvénient.

Si un nouveau type de pompe est installé ultérieurement, vous devez modifier la méthode **SwitchOnAllPumps** pour intégrer la nouvelle pompe. Cela signifie aussi que l'intégralité du code doit être totalement retestée, avec tout le temps d'inactivité et les coûts qui en découlent, car il s'agit d'une partie cruciale du logiciel.

Une solution

Pour résoudre ce problème, vous pouvez utiliser un mécanisme appelé *délégué*. La méthode **SwitchOnAllPumps** peut utiliser le délégué pour appeler la méthode appropriée afin d'activer une pompe sans avoir à déterminer son type.

Création de délégués



Un délégué contient une référence à une méthode plutôt qu'à son nom. À l'aide de délégués, vous pouvez appeler une méthode sans connaître son nom. L'appel du délégué exécute réellement la méthode à laquelle il fait référence.

Dans l'exemple de la centrale électrique, plutôt que de contenir des objets pompe, une **ArrayList** peut être utilisée pour contenir les délégués qui font référence aux méthodes nécessaires au démarrage de chaque pompe.

Un délégué est semblable à une interface. Il spécifie un contrat entre un appelant et un implémenteur. Un délégué associe un nom à la spécification d'une méthode. Une implémentation de la méthode peut être associée à ce nom, et un composant peut appeler la méthode à l'aide de ce nom. L'exigence principale de l'implémentation des méthodes est qu'elles doivent toutes posséder la même signature et retourner le même type de paramètres. Dans le cas du scénario de la centrale électrique, les méthodes **StartElectricPumpRunning** et **SwitchOn** ont toutes les deux la valeur **void**, et aucune n'accepte de paramètre.

Pour utiliser un délégué, vous devez d'abord le définir, puis l'instancier.

Définition de délégués

Un délégué spécifie le type de retour et les paramètres que chaque méthode doit fournir. La syntaxe à utiliser pour définir un délégué est la suivante :

```
public delegate void StartPumpCallback( );
```

Notez que la syntaxe de définition d'un délégué est semblable à celle de la définition d'une méthode. Dans cet exemple, vous définissez le délégué **StartPumpCallback** comme destiné à une méthode qui ne retourne aucune valeur (**void**) et n'accepte aucun paramètre. Cela correspond aux spécifications des méthodes **StartElectricPumpRunning** et **SwitchOn** dans les deux classes de pilotes de pompe.

Instanciation de délégués

Lorsque vous avez défini un délégué, vous devez l'instancier et le configurer pour qu'il fasse référence à une méthode. Pour instancier un délégué, utilisez son constructeur et fournissez la méthode d'objet qu'il doit invoquer lorsqu'il est appelé. Dans l'exemple suivant, une valeur **ElectricPumpDriver**, **ed1**, est créée, puis un délégué, **callback**, qui fait référence à la méthode **StartElectricPumpRunning** de **ed1**, est instancié :

```
public delegate void StartPumpCallback( );

void Example()
{
    ElectricPumpDriver ed1 = new ElectricPumpDriver( );

    StartPumpCallback callback;
    callback =
        ↪new StartPumpCallback(ed1.StartElectricPumpRunning);
    ...
}
```

Utilisation de délégués

- Pour appeler un délégué, utilisez la syntaxe d'une méthode

Aucun corps de méthode

```
public delegate void StartPumpCallback( );  
...  
StartPumpCallback callback;  
...  
callback = new  
    ↪ StartPumpCallback(ed1.StartElectricPumpRunning);  
...  
callback( );
```

Aucun appel ici

Appel ici

Un délégué est une variable qui appelle une méthode. Vous l'appellez de la même manière que s'il s'agissait d'une méthode, à la différence que le délégué remplace le nom de la méthode.

Exemple

Le code suivant montre comment définir, créer et appeler des délégués destinés à être utilisés par la centrale électrique. Il remplit l'**ArrayList** appelée *callbacks* avec des instances de délégués qui font référence aux méthodes utilisées pour actionner chaque pompe. La méthode **SwitchOnAllPumps** parcourt cette **ArrayList** et appelle chaque délégué successivement. Avec les délégués, la méthode n'a pas besoin d'effectuer de vérification de type. Cette solution est bien plus simple que la précédente.

```
public delegate void StartPumpCallback( );

public class CoreTempMonitor2
{
    public void Add(StartPumpCallback callback)
    {
        callbacks.Add(callback);
    }

    public void SwitchOnAllPumps( )
    {
        foreach(StartPumpCallback callback in callbacks)
        {
            callback( );
        }
    }

    private ArrayList callbacks = new ArrayList( );
}

class ExampleOfUse
{
    public static void Main( )
    {
        CoreTempMonitor2 ctm = new CoreTempMonitor2( );

        ElectricPumpDriver ed1 = new ElectricPumpDriver( );
        ctm.Add(
            new StartPumpCallback(ed1.StartElectricPumpRunning)
        );

        PneumaticPumpDriver pd1 = new PneumaticPumpDriver( );
        ctm.Add(
            new StartPumpCallback(pd1.SwitchOn)
        );

        ctm.SwitchOnAllPumps( );
    }
}
```

◆ Définition et utilisation d'événements

- **Fonctionnement des événements**
- **Définition d'événements**
- **Passage de paramètres d'événement**
- **Démonstration : Gestion d'événements**

Dans l'exemple de la centrale électrique, vous avez appris à utiliser un délégué pour résoudre le problème de l'actionnement de différents types de pompes de manière générique. Cependant, le composant qui contrôle la température du noyau du réacteur est toujours chargé d'avertir successivement chacune des pompes qu'elle doit s'actionner. Vous pouvez régler ce problème de notification à l'aide d'événements.

Les événements permettent à un objet d'avertir d'autres objets qu'une modification s'est produite. Les autres objets peuvent marquer leur intérêt pour un événement, et ils sont avertis lorsqu'il se produit.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- définir des événements ;
- gérer des événements.

Fonctionnement des événements

■ Éditeur

- Déclenche un événement pour avertir tous les objets intéressés (les abonnés)

■ Abonné

- Fournit une méthode à appeler lorsque l'événement est déclenché

Les événements permettent aux objets de *marquer leur intérêt* pour la modification d'autres objets. En d'autres termes, les événements permettent aux objets de s'inscrire pour être avertis des modifications apportées à d'autres objets. Ils utilisent le modèle éditeur/abonné.

Éditeur

Un éditeur est un objet qui gère son état interne. Cependant, lorsque son état est modifié, il peut déclencher un événement pour alerter les autres objets intéressés par sa modification.

Abonné

Un abonné est un objet qui marque son intérêt pour un événement. Il est alerté lorsqu'un éditeur déclenche l'événement. Un événement peut posséder zéro ou plusieurs abonnés.

Les événements peuvent être assez complexes. Pour faciliter leur compréhension et leur gestion, vous devez respecter certains principes lors de leur utilisation.

Définition d'événements

■ Définition d'un événement

```
public delegate void StartPumpCallback( );  
private event StartPumpCallback CoreOverheating;
```

■ Abonnement à un événement

```
PneumaticPumpDriver pd1 = new PneumaticPumpDriver( );  
...  
CoreOverheating += new StartPumpCallback(pd1.SwitchOn);
```

■ Notification d'un événement aux abonnés

```
public void SwitchOnAllPumps( ) {  
    if (CoreOverheating != null) {  
        CoreOverheating( );  
    }  
}
```

En C#, les événements utilisent des délégués pour appeler des méthodes afin qu'elles s'abonnent à des objets. Il sont multicasts. Cela signifie que lorsqu'un éditeur déclenche un événement, il peut entraîner l'appel de plusieurs délégués. Cependant, vous ne pouvez pas vous fier à l'ordre dans lequel les délégués sont appelés. Si un délégué lève une exception, il peut arrêter complètement le traitement de l'événement, et plus aucun autre délégué n'est appelé.

Définition d'un événement

Pour définir un événement, un éditeur définit d'abord un délégué sur lequel baser l'événement. Le code suivant définit un délégué appelé **StartPumpCallback** et un événement appelé **CoreOverheating** qui appelle le délégué **StartPumpCallback** lorsqu'il est déclenché :

```
public delegate void StartPumpCallback( );  
private event StartPumpCallback CoreOverheating;
```

Abonnement à un événement

Les objets qui s'abonnent spécifient une méthode à appeler lorsque l'événement est déclenché. Si l'événement n'a pas encore été instancié, les objets qui s'abonnent spécifient un délégué qui fait référence à la méthode lors de la création de l'événement. Si l'événement existe, les objets abonnés ajoutent un délégué qui appelle une méthode lorsque l'événement est déclenché.

Par exemple, dans le scénario de la centrale électrique, vous pouvez créer deux pilotes de pompe et les abonner à l'événement **CoreOverheating** :

```
ElectricPumpDriver ed1 = new ElectricPumpDriver( );
PneumaticPumpDriver pd1 = new PneumaticPumpDriver( );
...
CoreOverheating = new StartPumpCallback(
    ↪ed1.StartElectricPumpRunning);
CoreOverheating += new StartPumpCallback(pd1.SwitchOn);
```

Remarque Vous devez déclarer comme **void** les délégués (et méthodes) utilisés pour s'abonner à un événement. Cette restriction ne s'applique pas lorsqu'un délégué est utilisé sans événement.

Notification d'un événement aux abonnés

Pour avertir les abonnés, vous devez *déclencher* l'événement. La syntaxe utilisée est identique à celle d'un appel de méthode ou de délégué. Dans l'exemple de la centrale électrique, la méthode **SwitchOnAllPumps** du composant de contrôle de la température du noyau n'a plus besoin de parcourir de liste de délégués :

```
public void SwitchOnAllPumps( )
{
    if (CoreOverheating != null) {
        CoreOverheating( );
    }
}
```

Exécuter l'événement de cette manière entraîne l'appel de tous les délégués et, dans cet exemple, toutes les pompes qui s'abonnent à l'événement sont activées. Notez que le code vérifie d'abord que l'événement possède au moins un délégué abonné. Sans cette vérification, le code lèverait une exception en cas d'absence d'abonné.

Pour plus d'informations sur les principes et les méthodes conseillées à respecter lors de l'utilisation d'événements, recherchez « événements, indications d'utilisation » dans l'aide du Kit de développement .NET Framework SDK.

Passage de paramètres d'événement

- **Les paramètres d'événement doivent être passés comme EventArgs**
 - Définissez une classe qui descend de EventArgs pour servir de conteneur aux paramètres d'événement
- **La même méthode abonnée peut être appelée par plusieurs événements**
 - Passez toujours à la méthode l'événement éditeur (sender) comme premier paramètre

Principes des paramètres d'événement

Pour passer des paramètres à une méthode abonnée, insérez-les dans une classe unique qui fournit les accesseurs permettant de les extraire. Dérivez cette classe de **System.EventArgs**.

Par exemple, dans le scénario de la centrale électrique, supposons que les méthodes qui actionnent les pompes, **StartElectricPumpRunning** et **SwitchOn**, ont besoin de la température actuelle du noyau pour déterminer la vitesse à laquelle les pompes doivent fonctionner. Pour régler ce problème, vous devez créer la classe suivante afin de passer la température du noyau du composant de contrôle du noyau aux objets pompe :

```
public class CoreOverheatingEventArgs: EventArgs
{
    private readonly int temperature;

    public CoreOverheatingEventArgs(int temperature)
    {
        this.temperature = temperature;
    }

    public int GetTemperature( )
    {
        return temperature;
    }
}
```

La classe **CoreOverheatingEventArgs** contient un paramètre de type entier. Le constructeur stocke la température en interne et vous devez l'extraire au moyen de la méthode **GetTemperature**.

Objet sender

Un objet peut s'abonner auprès de plusieurs événements de différents éditeurs et utiliser la même méthode dans chaque cas. Par conséquent, il est d'usage pour un événement de passer aux abonnés des informations sur l'éditeur qui les a déclenchés. Par convention, il s'agit du premier paramètre passé à la méthode abonnée, qui s'appelle généralement *sender*. Le code suivant indique les nouvelles versions des méthodes **StartElectricPumpRunning** et **SwitchOn**, modifiées dans l'optique où *sender* est le premier paramètre et la température, le deuxième :

```
public class ElectricPumpDriver
{
    ...

    public void StartElectricPumpRunning(object sender,
    ↪CoreOverheatingEventArgs args)
    {
        // Examine la température
        int currentTemperature = args.GetTemperature( );

        // Actionne la pompe à la vitesse requise pour
        // cette température
        ...
    }
    ...
}

public class PneumaticPumpDriver
{
    ...

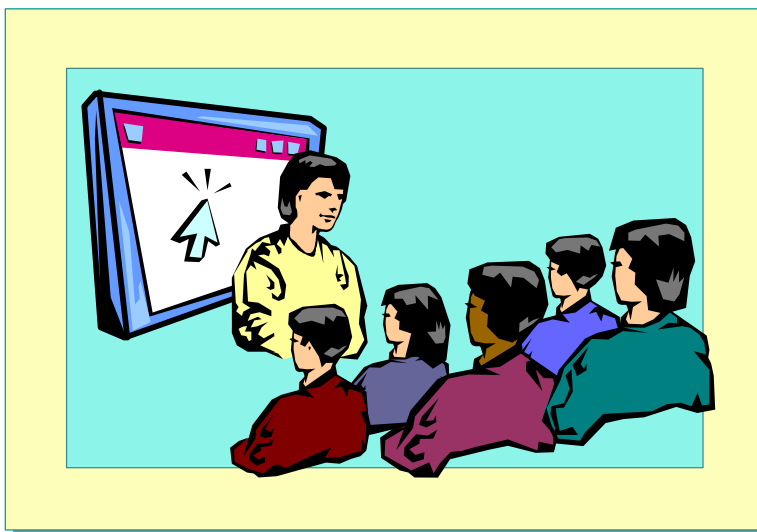
    public void SwitchOn(object sender,
    ↪CoreOverheatingEventArgs args)
    {
        // Examine la température
        int currentTemperature = args.GetTemperature( );

        // Actionne la pompe à la vitesse requise pour
        // cette température
        ...
    }
    ...
}
```

Remarque Vous devez également modifier le délégué dans le composant de contrôle de la température du noyau. Dans l'exemple de la centrale électrique, le délégué devient :

```
public delegate void StartPumpCallback(object sender,
    ↪CoreOverheatingEventArgs args);
```

Démonstration : Gestion d'événements



Dans cette démonstration, vous allez étudier à l'aide d'un exemple la manière dont vous pouvez utiliser des événements pour communiquer des informations entre objets.

Atelier 12.2 : Définition et utilisation d'événements



Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- publier des événements ;
- vous abonner à des événements ;
- passer des paramètres à des événements.

Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les éléments suivants :

- Création de classes en C#
- Définition de constructeurs et de destructeurs
- Compilation et utilisation d'assemblies

Durée approximative de cet atelier : 30 minutes

Exercice 1

Audit de transactions bancaires

Cet exercice développe l'exemple de la banque utilisé dans les ateliers précédents. Dans cet exercice, vous allez créer une classe appelée **Audit** dont l'objectif est d'enregistrer dans un fichier texte les modifications apportées aux soldes du compte bancaire. Le compte est ensuite averti des modifications par un événement publié par la classe **BankAccount**.

Vous utilisez les méthodes **Deposit** et **Withdraw** de la classe **BankAccount** pour déclencher l'événement, appelé **Auditing**, auquel un objet **Audit** est abonné.

L'événement **Auditing** accepte un paramètre contenant un objet **BankTransaction**. Si vous avez réalisé les ateliers précédents, vous devez vous rappeler que la classe **BankTransaction** contient les détails d'une transaction, tels que son montant, sa date de création, etc. Un objet **BankTransaction** est créé à chaque dépôt ou retrait effectué par l'utilisation d'un **BankAccount**.

Vous devez respecter l'ensemble des principes de gestion des événements mentionnés dans ce module.

► Pour définir la classe des paramètres d'événement

Dans cet exercice, un objet **BankTransaction** sera passé comme paramètre à l'événement déclenché. Les paramètres d'événement doivent être dérivés de **System.EventArgs**, afin qu'une nouvelle classe contenant un objet **BankTransaction** soit créée.

1. Ouvrez le projet Audit.sln situé dans le dossier *dossier d'installation\Labs\Lab12\Starter\Audit*.
2. Créez une nouvelle classe à l'aide de l'option **Ajouter un nouvel élément** du menu **Projet**. Veillez à créer une **Nouvelle classe C#** et appelez-la **AuditEventArgs.cs**.
3. Remplacez l'espace de noms par **Banking**.
4. Modifiez la définition de la classe **AuditEventArgs** afin qu'elle soit dérivée de **System.EventArgs**, comme suit :

```
public class AuditEventArgs : System.EventArgs
{
    ...
}
```

5. Créez une variable **BankTransaction** privée readonly appelée *transData*, et initialisez-la à **null**, comme suit :
- ```
private readonly BankTransaction transData = null;
```
6. Modifiez le constructeur par défaut pour accepter un paramètre **BankTransaction** unique appelé *transaction* et affectez-lui **this.transData**. Le code du constructeur est le suivant :

```
public AuditEventArgs(BankTransaction transaction)
{
 this.transData = transaction;
}
```

7. Fournissez une méthode d'accessor publique appelée **getTransaction** qui retourne la valeur de **this.transData**, comme suit :

```
public BankTransaction getTransaction()
{
 return this.transData;
}
```

8. Compilez le projet et corrigez les erreurs, le cas échéant.

► **Pour définir la classe Audit**

1. Dans le projet **Audit**, créez une classe à l'aide de l'option **Ajouter un nouvel élément** du menu **Projet**. Veillez à créer une **Nouvelle classe C#** et appelez-la **Audit.cs**. Il s'agit de la classe qui s'abonnera à l'événement **Auditing** et écrira les détails des transactions dans un fichier sur disque.
2. Une fois la classe créée, ajoutez un commentaire récapitulant l'objectif de la classe **Audit**. Aidez-vous de la description de l'exercice.
3. Remplacez l'espace de noms par **Banking**.
4. Ajoutez une directive **using** qui fait référence à **System.IO**.
5. Ajoutez une variable **string** privée appelée *filename* à la classe **Audit**.
6. Ajoutez une variable **StreamWriter** privée appelée *auditFile* à la classe **Audit**.

---

**Remarque** Une variable **StreamWriter** vous permet d'écrire des données dans un fichier. Dans l'atelier 6, vous avez utilisé **StreamReader** pour lire un fichier. Dans cet exercice, vous utilisez la méthode **AppendText** de la classe **StreamWriter**.

La méthode **AppendText** ouvre un fichier nommé pour y ajouter du texte. Elle écrit les données à la fin du fichier. La méthode **WriteLine** permet d'écrire réellement des données dans le fichier une fois qu'il est ouvert (comme la classe **Console**).

---

7. Modifiez le constructeur par défaut dans la classe **Audit** pour accepter un paramètre de type chaîne unique appelé *fileToUse*. Dans le constructeur :
- Affectez à **this.filename** le paramètre *fileToUse*.
  - Ouvrez ce fichier nommé en mode **AppendText** et stockez le descripteur du fichier dans *auditFile*.

Le code complet du constructeur est le suivant :

```
private string filename;
private StreamWriter auditFile;

public Audit(string fileToUse)
{
 this.filename = fileToUse;
 this.auditFile = File.AppendText(fileToUse);
}
```



8. Dans la classe **Audit**, ajoutez la méthode qui sera utilisée pour s'abonner à l'événement **Auditing** de la classe **BankTransaction**. Elle sera exécutée lorsqu'un objet **BankTransaction** déclenchera l'événement. Cette méthode doit être une méthode **void** publique appelée **RecordTransaction**. Elle accepte deux paramètres : un paramètre **object** appelé *sender* et un paramètre **AuditEventArgs** appelé *eventData*.
9. Dans la méthode **RecordTransaction**, ajoutez du code pour :
  - Créer une variable **BankTransaction** appelée *tempTrans*.
  - Exécuter **eventData.getTransaction( )** et assigner le résultat à *tempTrans*.
  - Si *tempTrans* n'est pas **null**, utiliser la méthode **WriteLine** de **this.auditFile** pour ajouter le montant de *tempTrans* (utilisez la méthode **Amount( )**) et la date de création (utilisez la méthode **When( )**) à la fin du fichier d'audit. Ne fermez pas le fichier.

---

**Remarque** Le paramètre *sender* n'est pas utilisé par cette méthode, mais par convention, toutes les méthodes de gestion des événements attendent l'expéditeur de l'événement comme premier paramètre.

---

Le code complet de cette méthode est le suivant :

```
public void RecordTransaction(object sender,
 AuditEventArgs eventData)
{
 BankTransaction tempTrans = eventData.getTransaction();
 if (tempTrans != null)
 this.auditFile.WriteLine("Montant : {0}\tDate :
{1}",
 tempTrans.Amount(), tempTrans.When());
}
```

10. Ajoutez une variable booléenne privée appelée **closed** à la classe **Audit** et initialisez-la à **false**.
11. Dans la classe **Audit**, créez une méthode **void Close** publique qui ferme **this.auditFile**. Le code de la méthode **Close** est le suivant :

```
public void Close()
{
 if (!closed)
 {
 this.auditFile.Close();
 closed = true;
 }
}
```

12. Compilez le projet et corrigez les erreurs, le cas échéant.

► **Pour tester la classe Audit**

1. Ouvrez le projet AuditTestHarness.sln situé dans le dossier *dossier d'installation\Labs\Lab12\Starter\AuditTestHarness*.
2. Exécutez les étapes suivantes pour ajouter une référence à la bibliothèque contenant votre classe **Audit** compilée. Il s'agit d'une bibliothèque de liaison dynamique (DLL, *Dynamic-Link Library*) appelée Audit.dll figurant dans le dossier *dossier d'installation\Labs\Lab12\Starter\Audit\bin\Debug*.
  - a. Dans l'Explorateur de solutions, développez l'arborescence du projet **AuditTestHarness**.
  - b. Cliquez avec le bouton droit sur **Références**.
  - c. Cliquez sur **Ajouter une référence**.
  - d. Cliquez sur **Parcourir**.
  - e. Accédez au dossier *dossier d'installation\Labs\Lab12\Starter\Audit\bin\Debug*.
  - f. Cliquez sur **Audit.dll**.
  - g. Cliquez sur **Ouvrir**, puis sur **OK**.
3. Dans la classe **Test**, examinez la méthode **Main**. Cette classe :
  - a. Crée une instance de la classe **Audit**, en utilisant le nom AuditTrail.dat comme nom du fichier de stockage des informations d'audit.
  - b. Crée un nouvel objet **BankTransaction** pour un montant de 500 dollars.
  - c. Crée un objet **AuditEventArgs** qui utilise l'objet **BankTransaction**.
  - d. Appelle la méthode **RecordTransaction** de l'objet **Audit**.

Le test est répété avec une deuxième transaction de –200 dollars.

Après le deuxième test, la méthode **Close** est appelée pour s'assurer que les enregistrements d'audit sont stockés sur le disque.
4. Compilez le projet.
5. Ouvrez une fenêtre d'invite de commandes et accédez au dossier *dossier d'installation\Labs\Lab12\Starter\AuditTestHarness\bin\Debug*. Ce dossier contient les fichiers AuditTestHarness.exe et Audit.dll. Il contient aussi les fichiers AuditTestHarness.pdb et Audit.pdb, que vous pouvez ignorer.
6. Exécutez **AuditTestHarness**.
7. À l'aide d'un éditeur de texte de votre choix (par exemple, WordPad), accédez au dossier *dossier d'installation\Labs\Lab12\Starter\AuditTestHarness\bin\Debug* et examinez le contenu du fichier AuditTrail.dat. Il doit contenir les données relatives aux deux transactions.

► Pour définir l'événement **Auditing**

1. Ouvrez le projet `Audit.sln` situé dans le dossier *dossier d'installation*\Labs\Lab12\Starter\Audit.
2. Dans le fichier `BankAccount.cs`, au-dessus de la classe **BankAccount**, déclarez un délégué public de type **void** qui s'appelle **AuditEventHandler** et accepte deux paramètres (un paramètre **Object** appelé *sender* et un paramètre **AuditEventArgs** appelé *data*), comme suit :

```
public delegate void AuditEventHandler(Object sender,
 ↪AuditEventArgs data);
```

3. Dans la classe **BankAccount**, déclarez un événement privé de type **AuditEventHandler** appelé **AuditingTransaction** et initialisez-le en lui affectant la valeur **null**, comme suit :

```
private event AuditEventHandler AuditingTransaction =
 ↪null;
```

4. Ajoutez une méthode **void** publique appelée **AddOnAuditingTransaction**. Elle accepte un paramètre **AuditEventHandler** unique appelé *handler* et a pour objectif d'ajouter *handler* à la liste des délégués qui s'abonnent à l'événement **AuditingTransaction**. La méthode aura l'aspect suivant :

```
public void AddOnAuditingTransaction(AuditEventHandler
 ↪handler)
{
 this.AuditingTransaction += handler;
}
```

5. Ajoutez une autre méthode **void** publique appelée **RemoveOnAuditingTransaction**. Elle accepte aussi un paramètre **AuditEventHandler** unique appelé *handler* et a pour objectif de supprimer *handler* de la liste des délégués qui s'abonnent à l'événement **AuditingTransaction**. La méthode aura l'aspect suivant :

```
public void RemoveOnAuditingTransaction(AuditEventHandler
 ↪handler)
{
 this.AuditingTransaction -= handler;
}
```

6. Ajoutez une troisième méthode que l'objet **BankAccount** utilisera pour déclencher l'événement et avertir tous les abonnés. Cette méthode doit être une méthode **void** protégée appelée **OnAuditingTransaction**. Elle accepte un paramètre **BankTransaction** appelé *bankTrans*. La méthode examine l'événement **this.AuditingTransaction**. S'il contient des délégués, elle crée un objet **AuditEventArgs** appelé *auditTrans*, qui est construit à l'aide du paramètre *bankTrans*. Ensuite, elle provoque l'exécution des délégués en se passant elle-même en tant qu'expéditeur de l'événement et en passant le paramètre *auditTrans* comme données. Le code de cette méthode est le suivant :

```
protected void OnAuditingTransaction(BankTransaction
bankTrans)
{
 if (this.AuditingTransaction != null) {
 AuditEventArgs auditTrans = new
 AuditEventArgs(bankTrans);
 this.AuditingTransaction(this, auditTrans);
 }
}
```

7. Dans la méthode **Withdraw** de **BankAccount**, ajoutez une instruction qui appelle **OnAuditingTransaction**. Passez l'objet de transaction créé par la méthode **Withdraw**. Cette instruction doit être insérée juste avant l'instruction **return** située à la fin de la méthode. Le code complet de la méthode **Withdraw** est le suivant :

```
public bool Withdraw(decimal amount)
{
 bool sufficientFunds = accBal >= amount;
 if (sufficientFunds) {
 accBal -= amount;
 BankTransaction tran =
 ↪new BankTransaction(-amount);
 tranQueue.Enqueue(tran);
 this.OnAuditingTransaction(tran);
 }
 return sufficientFunds;
}
```

8. Ajoutez une instruction similaire à la méthode **Deposit**. Le code complet de la méthode **Deposit** est le suivant :

```
public decimal Deposit(decimal amount)
{
 accBal += amount;
 BankTransaction tran = new BankTransaction(amount);
 tranQueue.Enqueue(tran);
 this.OnAuditingTransaction(tran);
 return accBal;
}
```

9. Compilez le projet et corrigez les erreurs, le cas échéant.

### ► Pour s'abonner à l'événement Auditing

1. La phase finale consiste à créer l'objet **Audit** qui doit s'abonner à l'événement **Auditing**. Cet objet **Audit** appartient à la classe **BankAccount** et est créé lors de l'instanciation de cette classe. De cette manière, chaque compte obtient sa propre liste d'audit.

Définissez une variable **Audit** privée appelée *accountAudit* dans la classe **BankAccount**, comme suit :

```
private Audit accountAudit;
```

2. Ajoutez à la classe **BankAccount** une méthode **void** publique appelée **AuditTrail**. Cette méthode crée un objet **Audit** et s'abonne à l'événement **Auditing**. Elle accepte un paramètre **string**, qui est le nom d'un fichier à utiliser pour la liste d'audit. Cette méthode effectue les opérations suivantes :
  - Elle instancie *accountAudit* à l'aide de ce paramètre **string**.
  - Elle crée une variable **AuditEventHandler** appelée *doAuditing* et l'instancie à l'aide de la méthode **RecordTransaction** de *accountAudit*.
  - Elle ajoute *doAuditing* à la liste des abonnés à l'événement **Auditing**. Elle utilise la méthode **AddOnAuditingTransaction** en passant *doAuditing* comme paramètre.

Le code complet de cette méthode est le suivant :

```
public void AuditTrail(string auditFileName)
{
 this.accountAudit = new Audit(auditFileName);
 AuditEventHandler doAuditing = new
 ↪AuditEventHandler(this.accountAudit.RecordTransaction);
 this.AddOnAuditingTransaction(doAuditing);
}
```

3. Dans le destructeur de la classe **BankAccount**, ajoutez une instruction qui appelle la méthode **Dispose**. (Cela permet de s'assurer que tous les enregistrements d'audit sont correctement écrits sur le disque.)
4. Dans la méthode **Dispose** de la classe **BankAccount**, ajoutez la ligne de code suivante au sein de l'instruction **if** :

```
accountAudit.Close();
```
5. Compilez le projet et corrigez les erreurs, le cas échéant.

► **Pour tester l'événement Auditing**

1. Ouvrez le projet EventTestHarness.sln situé dans le dossier *dossier d'installation\Labs\Lab12\Starter\EventTestHarness*.
2. Exécutez les étapes suivantes pour ajouter une référence à la DLL (Audit.dll) contenant vos classes **Audit** et **BankAccount** compilées. La bibliothèque Audit.dll se situe dans le dossier *dossier d'installation\Labs\Lab12\Starter\Audit\bin\Debug*.
  - a. Dans l'Explorateur de solutions, développez l'arborescence du projet **EventTestHarness**.
  - b. Cliquez avec le bouton droit sur **Références**.
  - c. Cliquez sur **Ajouter une référence**.
  - d. Cliquez sur **Parcourir**.
  - e. Accédez au dossier *dossier d'installation\Labs\Lab12\Starter\Audit\bin\Debug*.
  - f. Cliquez sur **Audit.dll**.
  - g. Cliquez sur **Ouvrir**, puis sur **OK**.
3. Dans la classe **Test**, examinez la méthode **Main**. Cette classe effectue les opérations suivantes :
  - a. Elle crée deux comptes bancaires.
  - b. Elle utilise la méthode **AuditTrail** pour provoquer la création des objets **Audit** incorporés dans chaque compte et leur abonnement à l'événement **Auditing**.
  - c. Elle exécute une série de dépôts et de retraits sur chaque compte.
  - d. Elle ferme les deux comptes.
4. Compilez le projet et corrigez les erreurs, le cas échéant.
5. Ouvrez une fenêtre d'invite de commandes et accédez au dossier *dossier d'installation\Labs\Lab12\Starter\EventTestHarness\bin\Debug*. Ce dossier contient les fichiers EventTestHarness.exe et Audit.dll. Il contient aussi les fichiers EventTestHarness.pdb et Audit.pdb, que vous pouvez ignorer.
6. Exécutez EventTestHarness.
7. À l'aide d'un éditeur de texte de votre choix, examinez le contenu des fichiers Account1.dat et Account2.dat. Ils doivent contenir les données relatives aux transactions effectuées sur les deux comptes.

## Contrôle des acquis

- Introduction aux opérateurs
- Surcharge d'opérateurs
- Création et utilisation de délégués
- Définition et utilisation d'événements

1. Les opérateurs arithmétiques d'assignation composés ( $+=$ ,  $-=$ ,  $*=$ ,  $/=$  et  $\%=$ ) peuvent-ils être surchargés ?
2. Dans quelles circonstances un opérateur de conversion doit-il être explicite ?

3. Comment les opérateurs de conversion explicites sont-ils appelés ?
4. Qu'est-ce qu'un délégué ?
5. Comment s'abonner à un événement ?
6. Dans quel ordre les méthodes qui s'abonnent à un événement sont-elles appelées ? Toutes les méthodes qui s'abonnent à un événement sont-elles toujours exécutées ?



---

## Module 13 : Propriétés et indexeurs

### Table des matières

|                                                            |    |
|------------------------------------------------------------|----|
| Vue d'ensemble                                             | 1  |
| Utilisation des propriétés                                 | 2  |
| Utilisation des indexeurs                                  | 18 |
| Atelier 13.1 : Utilisation de propriétés<br>et d'indexeurs | 34 |
| Contrôle des acquis                                        | 43 |



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

# Vue d'ensemble

- Utilisation des propriétés
- Utilisation des indexeurs

Vous pouvez exposer les attributs nommés d'une classe à l'aide de champs ou de propriétés. Les champs sont implémentés comme variables membres dotées d'un accès privé. En C#, les propriétés apparaissent à l'utilisateur d'une classe comme des champs, mais elles font appel à des méthodes pour obtenir ou définir des valeurs.

C# propose une fonctionnalité d'indexeur permettant aux objets d'être indexés à la manière d'un tableau.

Dans ce module, vous allez apprendre à utiliser les propriétés et les indexeurs. Vous verrez comment permettre un accès de type champ à l'aide de propriétés et comment permettre un accès de type tableau à l'aide d'indexeurs.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- créer des propriétés pour encapsuler des données dans une classe ;
- définir des indexeurs permettant d'avoir accès à des classes à l'aide d'une notation de type tableau.

## ◆ Utilisation des propriétés

- Pourquoi utiliser des propriétés ?
- Utilisation des accesseurs
- Comparaison entre les propriétés et les champs
- Comparaison entre les propriétés et les méthodes
- Types des propriétés
- Exemple de propriété

---

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- utiliser des propriétés pour encapsuler des données dans une classe ;
- utiliser des propriétés pour accéder à des données dans une classe.

## Pourquoi utiliser des propriétés ?

### ■ Les propriétés offrent :

- un moyen utile d'encapsuler des informations au sein d'une classe ;
- syntaxe concise ;
- souplesse.

Les propriétés constituent un moyen utile d'encapsuler des données au sein d'une classe. Les propriétés décrivent par exemple la longueur d'une chaîne, la taille d'une police de caractère, la légende d'une fenêtre, le nom d'un client, etc.

### Syntaxe concise

C# ajoute les propriétés en tant qu'éléments de première classe du langage. Dans de nombreux langages existants, tels que Microsoft® Visual Basic®, les propriétés sont des éléments de première classe. Si vous considérez une propriété comme un champ, cela peut vous aider à appréhender la logique de l'application. Comparez, par exemple, les deux instructions suivantes : la première n'utilise pas de propriétés, alors que la seconde en utilise.

```
o.SetValue(o.GetValue() + 1);
```

```
o.Value++;
```

L'instruction ayant recours à une propriété est certainement plus facile à comprendre et moins source d'erreurs.

## Souplesse

Pour lire ou écrire la valeur d'une propriété, vous devez utiliser une syntaxe de type champ. (C'est-à-dire que vous n'utilisez pas de parenthèses). Cependant, le compilateur traduit cette syntaxe de type champ en accesseurs **get** et **set** encapsulés de type méthode. Par exemple, **Value** pourrait être une propriété de l'objet **o** dans l'expression **o.Value**, qui entraînerait l'exécution des instructions dans la « méthode » d'accesseur **get** pour la propriété **Value**. Cette séparation permet la modification des instructions dans les accesseurs **get** et **set** d'une propriété sans que cela n'affecte l'utilisation de la propriété, qui conserve sa syntaxe de type champ. Du fait de cette souplesse, il est préférable d'utiliser des propriétés plutôt que des champs lorsque cela est possible.

Lorsque vous exposez un état par le biais d'une propriété, votre code est potentiellement moins efficace que lorsque vous exposez l'état directement par le biais d'un champ. Cependant, lorsqu'une propriété contient seulement une petite quantité de code et qu'elle est non virtuelle (ce qui est souvent le cas), l'environnement d'exécution peut remplacer les appels à un accesseur par le code de l'accesseur. Ce processus est connu sous le nom de *fonctionnalité inline* ; il rend l'accès aux propriétés aussi efficace que l'accès aux champs tout en conservant la souplesse accrue des propriétés.

## Utilisation des accesseurs

### ■ Les propriétés fournissent un accès de type champ

- Utilisez les instructions de l'accesseur **get** pour fournir un accès en lecture
- Utilisez les instructions de l'accesseur **set** pour fournir un accès en écriture

```
class Button
{
 public string Caption // Propriété
 {
 get return caption;
 set caption = value; }
 private string caption; // Champ
}
```

Une propriété est un membre de classe proposant un accès au champ d'un objet. Elle permet d'associer des actions à la lecture et à l'écriture de l'attribut d'un objet. La déclaration d'une propriété est constituée d'un type et d'un nom, et possède une ou deux portions de code appelées des accesseurs. Ces accesseurs sont :

- l'accesseur **get** ;
- l'accesseur **set**.

Les accesseurs n'ont aucun paramètre. Une propriété n'a pas besoin d'avoir un accesseur **get** et un accesseur **set**. Par exemple, une propriété en lecture seule fournit seulement un accesseur **get**. Vous en apprendrez davantage sur les propriétés en lecture seule plus loin dans cette section.

### Utilisation de l'accesseur **get**

L'accesseur **get** d'une propriété retourne la valeur de cette propriété. Le code suivant fournit un exemple :

```
public string Caption
{
 get { return caption; }
 ...
}
```

Vous appelez implicitement l'accesseur **get** d'une propriété lorsque vous utilisez cette propriété dans un contexte de lecture. Le code suivant fournit un exemple :

```
Button myButton;
...
string cap = myButton.Caption; // Appelle "myButton.Caption.get"
```

Vous remarquerez que l'on n'utilise pas de parenthèses après le nom de la propriété. Dans cet exemple, l'instruction `return caption;` retourne une chaîne. Cette chaîne est retournée à chaque fois que la valeur de la propriété **Caption** est lue.

La lecture d'une propriété ne doit pas changer les données de l'objet. Lorsque vous appelez un accesseur **get**, cela équivaut conceptuellement à lire la valeur d'un champ. Un accesseur **get** ne doit pas avoir d'effets secondaires notables.

## Utilisation de l'accesseur set

L'accesseur **set** d'une propriété modifie la valeur de cette propriété.

```
public string Caption
{
 ...
 set { caption = value; }
}
```

Vous appelez implicitement l'accesseur **set** d'une propriété lorsque vous utilisez cette propriété dans un contexte d'écriture, à savoir lorsque vous l'utilisez dans une assignation. Le code suivant fournit un exemple :

```
Button myButton;
...
myButton.Caption = "OK"; // Appelle "myButton.Caption.set"
```

Vous remarquerez à nouveau l'absence de parenthèses. La variable *value* contient la valeur que vous assignez ; elle est créée automatiquement par le compilateur. Au sein de l'accesseur **set** de la propriété **Caption**, *value* peut être considéré comme une variable de type string contenant la chaîne « OK. » Un accesseur **set** ne peut pas retourner de valeur.

L'appel à un accesseur **set** est identique dans sa syntaxe à une simple assignation, vous devriez donc limiter ses effets secondaires notables. Par exemple, il serait quelque peu inattendu que l'instruction suivante modifie à la fois la vitesse (speed) et la couleur de l'objet **thing**.

```
thing.speed = 5;
```

Il arrive toutefois que les effets secondaires de l'accesseur **set** soient utiles. Par exemple, un panier de courses peut mettre à jour son total à chaque fois que le décompte d'articles change.



## Comparaison entre les propriétés et les champs

- **Les propriétés sont des « champs logiques »**
  - L'accesseur **get** peut retourner une valeur calculée
- **Similitudes**
  - La syntaxe de création et d'utilisation est la même
- **Différences**
  - Les propriétés ne sont pas des valeurs ; elles n'ont pas d'adresse
  - Les propriétés ne peuvent pas être utilisées en tant que paramètres **ref** ou **out** de méthodes

En tant que développeur expérimenté, vous savez déjà utiliser les champs. De par la similitude entre les champs et les propriétés, il est utile de comparer ces deux éléments de programmation.

### Les propriétés sont des champs logiques

Vous pouvez utiliser l'accesseur **get** d'une propriété pour calculer une valeur plutôt que de retourner directement la valeur d'un champ. Envisagez les propriétés comme des champs logiques, c'est-à-dire des champs n'ayant pas nécessairement une implémentation physique directe. Par exemple, une classe **Person** peut contenir un champ correspondant à la date de naissance de la personne et une propriété calculant l'âge de la personne :

```
Class Person
{
 public Person(DateTime born)
 {
 this.born = born;
 }

 public int Age
 {
 // Simplifiée...
 get { return DateTime.UtcNow.Year - born.Year; }
 }
 ...
 private readonly DateTime born;
}
```

## Similitudes avec les champs

Les propriétés sont une extension naturelle des champs. Comme les champs :

- Elles spécifient un nom avec un type non void associé, comme suit :

```
class Example
{
 int field;
 int Property { ... }
}
```

- Elles peuvent être déclarées avec n'importe quel modificateur d'accès, comme suit :

```
class Example
{
 private int field;
 public int Property { ... }
}
```

- Elles peuvent être statiques, comme suit :

```
class Example
{
 static private int field;
 static public int Property { ... }
}
```

- Elles peuvent masquer des membres de classe de base portant le même nom, comme suit :

```
class Base
{
 public int field;
 public int Property { ... }
}
class Example: Base
{
 public int field;
 new public int Property { ... }
}
```

- Elles sont assignées ou lues par le biais de la syntaxe de champ, comme suit :

```
Example o = new Example();
o.field = 42;
o.Property = 42;
```

## Différences entre les propriétés et les champs

À la différence des champs, les propriétés ne correspondent pas directement à des emplacements de stockage. Même si vous utilisez la même syntaxe pour accéder à une propriété que celle que vous utiliseriez pour accéder à un champ, cette propriété n'est pas considérée comme une variable. Vous ne pouvez donc pas passer une propriété en tant que paramètre **ref** ou **out** sans obtenir d'erreur de compilation. Le code suivant fournit un exemple :

```
class Example
{
 public string Property
 {
 get { ... }
 set { ... }
 }
 public string Field;
}
class Test
{
 static void Main()
 {
 Example eg = new Example();

 ByRef(ref eg.Property); // Erreur de compilation
 ByOut(out eg.Property); // Erreur de compilation

 ByRef(ref eg.Field); // OK
 ByOut(out eg.Field); // OK
 }
 static void ByRef(ref string name) { ... }
 static void ByOut(out string name) { ... }
}
```

## Comparaison entre les propriétés et les méthodes

### ■ Similitudes

- Toutes deux contiennent du code à exécuter
- Elles peuvent être utilisées pour masquer les informations relatives à l'implémentation
- Elles peuvent être marquées comme étant virtuelles, abstraites ou de substitution

### ■ Différences

- Syntaxiques les propriétés n'utilisent pas de parenthèses
- Sémantiques les propriétés ne peuvent pas être **void** ou accepter des paramètres arbitraires

---

## Similitudes avec les méthodes

Avec les propriétés et les méthodes, vous pouvez :

- spécifier des instructions à exécuter ;
- spécifier un type de retour devant être au moins aussi accessible que la propriété ;
- les marquer comme étant virtuelles, abstraites ou de substitution ;
- les introduire dans une interface ;
- fournir une séparation entre l'état interne d'un objet et son interface publique (ce que vous ne pouvez pas faire avec un champ).

Ce dernier point est sûrement le plus important. Vous pouvez modifier l'implémentation d'une propriété sans affecter la syntaxe avec laquelle vous l'utilisez. Par exemple, dans le code suivant, vous remarquerez que la propriété **TopLeft** de la classe **Label** est implémentée directement avec un champ **Point**.

```
struct Point
{
 public Point(int x, int y)
 {
 this.x = x;
 this.y = y;
 }
 public int x, y;
}
class Label
{
 ...
 public Point TopLeft
 {
 get { return topLeft; }
 set { topLeft = value; }
 }
 private Point topLeft;
}
class Use
{
 static void Main()
 {
 Label text = new Label(...);
 Point oldPosition = text.TopLeft;
 Point newPosition = new Point(10,10);
 text.TopLeft = newPosition;
 }
 ...
}
```

**TopLeft** étant implémentée en tant que propriété, vous pouvez également l'implémenter sans changer la syntaxe avec laquelle vous l'utilisez, comme le montre l'exemple suivant, qui utilise deux champs **int** nommés *x* et *y* plutôt que le champ **Point** nommé *topLeft* :

```
class Label
{
 public Point TopLeft
 {
 get { return new Point(x,y); }
 set { x = value.x; y = value.y; }
 }
 private int x, y;
}
class Use
{
 static void Main()
 {
 Label text = new Label(...);
 // Strictement identique
 Point oldPosition = text.TopLeft;
 Point newPosition = new Point(10,10);
 text.TopLeft = newPosition;
 ...
 }
}
```

## Différences entre les propriétés et les méthodes

Les propriétés et les méthodes diffèrent par quelques aspects importants, répertoriés dans le tableau ci-après.

| Fonctionnalité                          | Propriétés | Méthodes |
|-----------------------------------------|------------|----------|
| Utilisation des parenthèses             | Non        | Oui      |
| Spécification de paramètres arbitraires | Non        | Oui      |
| Utilisation du type void                | Non        | Oui      |

Examinez les exemples suivants :

- Les propriétés n'utilisent pas de parenthèses ; les méthodes oui.

```
class Example
{
 public int Property { ... }
 public int Method() { ... }
}
```

- Les propriétés ne peuvent pas spécifier de paramètres arbitraires ; les méthodes oui.

```
class Example
{
 public int Property { ... }
 public int Method(double d1, decimal d2) { ... }
}
```

- Les propriétés ne peuvent pas être de type **void** ; les méthodes oui.

```
class Example
{
 public void Property { ... } // Erreur de compilation
 public void Method() { ... } // OK
}
```

## Types des propriétés

- **Propriétés en lecture/écriture**
  - Elles disposent des accesseurs **get** et **set**
- **Propriétés en lecture seule**
  - Elles ne disposent que de l'accesseur **get**
  - Elles ne sont pas constantes
- **Propriétés en écriture seule usage très limité**
  - Elles ne disposent que de l'accesseur **set**
- **Propriétés statiques**
  - Elles s'appliquent à la classe et ne peuvent accéder qu'aux données statiques

---

Lorsque vous utilisez des propriétés, vous pouvez définir les opérations autorisées pour chacune d'entre elles. Les opérations sont définies comme suit :

- **Propriétés en lecture/écriture**

Lorsque vous implémentez **get** et **set**, vous avez un accès en lecture et en écriture à la propriété.
- **Propriétés en lecture seule**

Lorsque vous implémentez uniquement **get**, vous avez un accès en lecture seule à la propriété.
- **Propriétés en écriture seule**

Lorsque vous implémentez uniquement **set**, vous avez un accès en écriture seule à la propriété.

### Utilisation de propriétés en lecture seule

Les propriétés uniquement dotés d'un accesseur **get** s'appellent des propriétés en lecture seule. Dans l'exemple ci-après, la classe **BankAccount** possède une propriété **Balance** avec un accesseur **get**, mais pas d'accesseur **set**. Par conséquent, **Balance** est une propriété en lecture seule.

```
class BankAccount
{
 private decimal balance;
 public decimal Balance
 {
 get { return balance; } // Mais pas d'accesseur set
 }
}
```



Vous ne pouvez pas assigner de valeur à une propriété en lecture seule. Par exemple, si vous ajoutez les instructions ci-après à l'exemple précédent, vous obtiendrez une erreur de compilation.

```
BankAccount acc = new BankAccount();
acc.Balance = 1000000M; // Erreur de compilation
```

Une erreur courante consiste à penser qu'une propriété en lecture seule spécifie une valeur constante. Or, cela n'est pas le cas. Dans l'exemple suivant, la propriété **Balance** est en lecture seule, ce qui signifie que vous pouvez seulement lire la valeur du solde. La valeur de ce solde peut néanmoins changer au fil du temps. Par exemple, le solde augmente après un dépôt.

```
class BankAccount
{
 private decimal balance;
 public decimal Balance
 {
 get { return balance; }
 }
 public void Deposit(decimal amount)
 {
 balance += amount;
 }
 ...
}
```

## Utilisation de propriétés en écriture seule

Les propriétés uniquement dotées d'un accesseur **set** s'appellent des propriétés en écriture seule. En règle générale, évitez de les utiliser dans la mesure du possible.

Si une propriété ne possède pas d'accesseur **get**, il est impossible de lire sa valeur ; on peut seulement lui en assigner une. Si vous tentez de lire une propriété ne possédant pas d'accesseur **get**, vous obtiendrez une erreur de compilation.

## Propriétés statiques

Une propriété statique, à l'instar d'un champ ou d'une méthode statiques, est associée à une classe, pas à un objet. Puisqu'une propriété statique n'est pas associée à une instance spécifique, elle peut accéder uniquement à des données statiques et ne peut ni faire référence à **this**, ni instancier des données. Vous trouverez ci-dessous un exemple :

```
class MyClass
{
 private int MyData = 0;

 public static int ClassData
 {
 get {
 return this.MyData; // Erreur de compilation
 }
 }
}
```

Vous ne pouvez pas inclure de modificateur **virtual**, **abstract** ou **override** sur une propriété statique.

## Exemple de propriété

```
public class Console
{
 public static TextReader In
 {
 get {
 if (reader == null) {
 reader = new StreamReader(...);
 }
 return reader;
 }
 ...
 private static TextReader reader = null;
 }
}
```

### Création juste-à-temps

Vous pouvez faire appel à des propriétés pour retarder l'initialisation d'une ressource jusqu'au moment où elle est référencée pour la première fois. Cette technique est connue sous le nom de *création ou instantiation « paresseuse »* ou *création juste-à-temps*. Le code suivant montre un exemple de création juste-à-temps provenant du kit de développement Microsoft .NET Framework (simplifié et non thread-safe) :

```
public class Console
{
 public static TextReader In
 {
 get {
 if (reader == null) {
 reader = new StreamReader(...);
 }
 return reader;
 }
 ...
 private static TextReader reader = null;
 }
}
```

Dans le code, vous remarquerez que :

- le champ sous-jacent appelé *reader* est initialisé à **null** ;
- seul le premier accès en lecture exécute le corps de l'instruction **if** dans l'accesseur **get**, créant ainsi l'objet **new StreamReader**. (**StreamReader** est dérivé de **TextReader**.)

## ◆ Utilisation des indexeurs

- Qu'est ce qu'un indexeur ?
- Comparaison des indexeurs et des tableaux
- Comparaison des indexeurs et des propriétés
- Utilisation de paramètres pour définir des indexeurs
- Exemple : String
- Exemple : BitArray

---

Un *indexeur* est un membre qui permet à un objet d'être indexé de la même manière qu'un tableau. Alors que vous pouvez faire appel à des propriétés pour permettre un accès de type champ aux données de votre classe, vous pouvez utiliser des indexeurs pour permettre un accès de type tableau aux membres de votre classe.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- définir des indexeurs ;
- utiliser des indexeurs.

## Qu'est ce qu'un indexeur ?

- **Un indexeur fournit un accès de type tableau à un objet**
  - Utile si une propriété peut avoir plusieurs valeurs
- **Pour définir un indexeur**
  - Créez une propriété appelée *this*
  - Spécifiez le type d'index
- **Pour utiliser un indexeur**
  - Utilisez une notation de type tableau pour lire ou écrire la propriété indexée

Un objet est constitué d'un certain nombre de sous-éléments. (Par exemple, une zone de liste est constituée d'un certain nombre de chaînes.) Les indexeurs vous permettent d'avoir accès aux sous-éléments à l'aide d'une notation de type tableau.

### Définition d'indexeurs

Le code suivant montre comment implémenter un indexeur qui fournit l'accès à un tableau privé de chaînes appelé **list** :

```
class StringList
{
 private string[] list;

 public string this[int index]
 {
 get { return list[index]; }
 set { list[index] = value; }
 }
 ...
 // Autres code et constructeurs destinés à initialiser
 // list
}
```

L'indexeur est une propriété appelée **this**, signalée par des crochets, contenant le type d'index utilisé. (Les indexeurs doivent toujours s'appeler **this** ; ils ne portent jamais de noms qui leur sont propres. En effet, on y accède par l'objet auquel ils appartiennent.) Dans ce cas, l'indexeur a besoin d'un **int** pour identifier la valeur à retourner ou à modifier par les accesseurs.

## Utilisation d'indexeurs

Vous pouvez faire appel à l'indexeur de la classe **StringList** pour obtenir un accès en lecture et en écriture aux membres de **myList**, comme le montre le code suivant :

```
...
StringList myList = new StringList();
...
myList[3] = "o"; // Indexeur (écriture)
...
string myString = myList[3]; // Indexeur (lecture)
...
```

Vous remarquerez que la syntaxe pour la lecture ou l'écriture de l'indexeur est très proche de celle destinée à utiliser un tableau. Le fait de référencer **myList** à l'aide d'un **int** entre crochets entraîne l'utilisation de l'indexeur. L'accesseur **get** ou l'accesseur **set** sera appelé, selon si vous lisez ou écrivez l'indexeur.

## Comparaison des indexeurs et des tableaux

### ■ Similitudes

- Ils utilisent la notation de type tableau

### ■ Différences

- Les indexeurs peuvent utiliser des indices non-entiers
- Les indexeurs peuvent être surchargés vous pouvez définir plusieurs indexeurs utilisant un type d'index différent
- Les indexeurs ne sont pas des variables, ils ne désignent pas d'emplacement de stockage vous ne pouvez pas passer un indexeur en tant que paramètre **ref** ou **out**

Même si les indexeurs utilisent une notation de type tableau, il existe quelques différences importantes entre les indexeurs et les tableaux.

### Définition des types d'index

L'index permettant d'accéder à un tableau doit être de type entier. Vous pouvez définir des indexeurs de manière à ce qu'ils acceptent d'autres types d'index. Par exemple, le code suivant montre comment utiliser un indexeur de chaînes :

```
class NickNames
{
 private Hashtable names = new Hashtable();

 public string this[string realName]
 {
 get { return (string)names[realName]; }
 set { names[realName] = value; }
 }
 ...
}
```

Dans l'exemple suivant, la classe **NickNames** stocke des paires de noms et de surnoms. Vous pouvez stocker un surnom et l'associer à un nom, pour ensuite rechercher le surnom lié à un nom.

```
...
NickNames myNames = new NickNames();
...
myNames["Paul"] = "Popaul";
...
string myNickName = myNames["Paul"];
...
```

## Surcharge

Une classe peut posséder plusieurs indexeurs si ceux-ci utilisent différents types d'index. Vous pouvez étendre la classe **NickNames** pour créer un indexeur prenant un index d'entiers. L'indexeur peut parcourir par itération la table de hachage le nombre de fois spécifié et retourner la valeur qu'il y trouve. Vous trouverez ci-dessous un exemple :

```
class NickNames
{
 private Hashtable names = new Hashtable();

 public string this[string realName]
 {
 get { return (string)names[realName]; }
 set { names[realName] = value; }
 }

 public string this[int nameNumber]
 {
 get
 {
 string nameFound;
 // Code parcourant la table de hachage
 // et renseignant nameFound
 return nameFound;
 }
 }
 ...
}
```



## Les indexeurs ne sont pas des variables

À la différence des tableaux, les indexeurs ne correspondent pas directement aux emplacements de stockage. En effet, ils possèdent des accesseurs **get** et **set** spécifiant les instructions à exécuter afin de lire ou d'écrire leurs valeurs. Cela signifie que même si vous utilisez la même syntaxe pour accéder à un indexeur que celle que vous utilisez pour un tableau (des crochets dans les deux cas), un indexeur n'est pas classé comme une variable.

Si vous transmettez un indexeur en tant que paramètre **ref** ou **out**, vous obtenez des erreurs de compilation, comme dans l'exemple suivant :

```
class Example
{
 public string[] array;

 public string this[int index]
 {
 get { ... }
 set { ... }
 }
}

class Test
{
 static void Main()
 {
 Example eg = new Example();

 ByRef(ref eg[0]); // Erreur de compilation
 ByOut(out eg[0]); // Erreur de compilation

 ByRef(ref eg.array[0]); // OK
 ByOut(ref eg.array[0]); // OK
 }
 static void ByRef(ref string name) { ... }
 static void ByOut(out string name) { ... }
}
```

## Comparaison des indexeurs et des propriétés

### ■ Similitudes

- Ils utilisent des accesseurs **get** et **set**
- Ils n'ont pas d'adresse
- Ils ne peuvent pas être void

### ■ Différences

- Les indexeurs peuvent être surchargés
- Les indexeurs ne peuvent pas être statiques

---

Les indexeurs s'appuient sur des propriétés et partagent nombre de leurs fonctionnalités. Ils diffèrent également des propriétés par certains aspects. Pour pouvoir comprendre pleinement les indexeurs, il est utile de les comparer aux propriétés.

### Similitudes avec les propriétés

Les indexeurs ressemblent aux propriétés par de nombreux aspects.

- Tous deux utilisent des accesseurs **get** et **set**.
- Aucun des deux ne définit d'emplacement de stockage physique ; par conséquent, ils ne peuvent pas être utilisés en tant que paramètres **ref** ou **out**.

```
class Dictionary
{
 public string this[string index]
 {
 get { ... }
 set { ... }
 }
}
Dictionary oed = new Dictionary();
...
ByRef(ref oed["vie"]); // Erreur de compilation
ByOut(ref oed["vie"]); // Erreur de compilation
```

- Aucun des deux ne peut spécifier un type **void**.

Par exemple, dans le code ci-dessus, `oed["vie"]` est une expression de type **string** qui ne peut être une expression de type **void**.

## Différences entre les indexeurs et les propriétés

Il est également important de comprendre par quels aspects les indexeurs et les propriétés diffèrent :

- Identification

Une propriété est identifiée uniquement par son nom. Un indexeur est identifié par sa signature, à savoir, par les crochets et le type de paramètres d'indexation.

- Surcharge

Une propriété étant identifiée uniquement par son nom, elle ne peut être surchargée. Cependant, puisque la signature d'un indexeur contient les types de ses paramètres, un indexeur peut être surchargé.

- Statique ou dynamique

Une propriété peut être un membre statique, alors qu'un indexeur est toujours membre d'une instance.

## Utilisation de paramètres pour définir des indexeurs

### ■ Lorsque vous définissez des indexeurs

- Spécifiez au moins un paramètre d'indexeur
- Spécifiez une valeur pour chaque paramètre
- N'utilisez pas les modificateurs de paramètres **ref** ou **out**

---

Il existe trois règles que vous devez respecter pour définir des indexeurs :

- spécifier au moins un paramètre d'indexeur ;
- spécifier une valeur pour chaque paramètre ;
- ne pas utiliser les modificateurs de paramètres **ref** ou **out**.

### Règles syntaxiques pour les paramètres d'indexeurs

Lorsque vous définissez un indexeur, vous devez spécifier au moins un paramètre (index). Vous en avez déjà vu des exemples. Certaines restrictions pèsent sur la classe de stockage du paramètre. Par exemple, vous ne pouvez pas utiliser les modificateurs de paramètres **ref** et **out** :

```
class BadParameter
{
 // Erreur de compilation
 public string this[ref int index] { ... }
 public string this[out string index] { ... }
}
```

## Paramètres multiples

Vous pouvez spécifier plusieurs paramètres dans un indexeur. Le code suivant fournit un exemple :

```
class MultipleParameters
{
 public string this[int one, int two]
 {
 get { ... }
 set { ... }
 }
 ...
}
```

Pour pouvoir utiliser l'indexeur de la classe **MultipleParameters**, vous devez spécifier deux valeurs, comme le montre le code suivant :

```
...
MultipleParameters mp = new MultipleParameters();
string s = mp[2,3];
...
```

Il s'agit de l'équivalent pour l'indexeur d'un tableau à plusieurs dimensions.

## Exemple : String

### ■ La classe String

- Est une classe immuable
- Utilise un indexeur (accesseur **get**, mais pas d'accesseur **set**)

```
class String
{
 public char this[int index]
 {
 get {
 if (index < 0 || index >= Length)
 throw new IndexOutOfRangeException();
 ...
 }
 ...
 }
}
```

Le type **string** est primordial en C#. Il s'agit d'un mot clé qui est un alias pour la classe **System.String** de la même façon que **int** est un alias du struct **System.Int32**.

### La classe String

La classe **String** est une classe immuable, scellée. Cela signifie que lorsque vous appelez une méthode sur un objet **string**, vous êtes assuré que la méthode ne modifie pas cet objet **string**. Si une méthode **string** retourne une chaîne, il s'agira d'une nouvelle chaîne.

### La méthode Trim

Pour supprimer les espaces à la fin d'une chaîne de caractères, utilisez la méthode **Trim** :

```
public sealed class String
{
 ...
 public String Trim() { ... }
 ...
}
```

La méthode **Trim** retourne une nouvelle chaîne tronquée, mais la chaîne utilisée pour appeler **Trim** reste entière. Le code suivant fournit un exemple :

```
string s = " Tronquez-moi ";
string t = s.Trim();
Console.WriteLine(s); // Écrit " Tronquez-moi "
Console.WriteLine(t); // Écrit "Tronquez-moi"
```

## L'indexeur de la classe **String**

Aucune méthode de la classe **String** ne modifie jamais la chaîne permettant d'appeler la méthode. Vous définissez la valeur d'une chaîne au moment de sa création et cette valeur ne change jamais.

À cause de ce choix de conception, la classe **String** possède un indexeur déclaré avec un accesseur **get** mais sans accesseur **set**, comme dans l'exemple suivant :

```
class String
{
 public char this[int index]
 {
 get {
 if (index < 0 || index >= Length)
 throw new IndexOutOfRangeException();
 ...
 }
 }
 ...
}
```

Si vous tentez d'utiliser un indexeur de chaînes pour écrire dans la chaîne, vous obtenez une erreur de compilation :

```
string s = "Sharp";
Console.WriteLine(s[0]); // OK
s[0] = 'S'; // Erreur de compilation
s[4] = 'k'; // Erreur de compilation
```

La classe **String** possède une classe compagne appelée **StringBuilder** dotée d'un indexeur en lecture et écriture.

## Exemple : BitArray

```
class BitArray
{
 public bool this[int index]
 {
 get {
 BoundsCheck(index);
 return (bits[index >> 5] & (1 << index)) != 0;
 }
 set {
 BoundsCheck(index);
 if (value) {
 bits[index >> 5] |= (1 << index);
 } else {
 bits[index >> 5] &= ~(1 << index);
 }
 }
 }
 private int[] bits;
}
```

---

Il s'agit d'un exemple plus complexe d'indexeurs, s'appuyant sur la classe **BitArray** du kit de développement .NET Framework SDK. En implémentant des indexeurs, la classe **BitArray** utilise moins de mémoire que le tableau booléen correspondant.

### Comparaison entre la classe BitArray et le tableau booléen

L'exemple ci-dessous illustre la création d'un tableau d'indicateurs booléens.

```
bool[] flags = new bool[32];
flags[12] = false;
```

Ce code fonctionne, mais malheureusement il utilise un octet pour stocker chaque **bool**. L'état de l'indicateur booléen (**true** ou **false**) peut être stocké dans un seul bit, mais un octet contient huit bits. Par conséquent, un tableau de booléens utilise huit fois plus de mémoire qu'il n'en a besoin.



Pour résoudre ce problème de mémoire, le kit de développement .NET Framework SDK fournit la classe **BitArray**, qui implémente des indexeurs et utilise également moins de mémoire que le tableau booléen correspondant. Vous trouverez ci-dessous un exemple :

```
class BitArray
{
 public bool this[int index]
 {
 get {
 BoundsCheck(index);
 return (bits[index >> 5] & (1 << index)) != 0;
 }
 set {
 BoundsCheck(index);
 if (value) {
 bits[index >> 5] |= (1 << index);
 } else {
 bits[index >> 5] &= ~(1 << index);
 }
 }
 }
 ...
 private int[] bits;
}
```

## Fonctionnement de BitArray

Pour comprendre le fonctionnement de la classe **BitArray**, voyez ce que fait le code à chaque étape :

1. Il stocke 32 **bools** dans un **int**.

**BitArray** utilise beaucoup moins de mémoire qu'un tableau booléen correspondant en stockant l'état de 32 **bools** dans un seul **int**. (N'oubliez pas que **int** est un alias de **Int32**.)

2. Il implémente un indexeur :

```
public bool this[int index]
```

La classe **BitArray** contient un indexeur permettant d'utiliser un objet **BitArray** comme un tableau. En fait, vous pouvez utiliser un **BitArray** exactement comme un **bool** [ ].

```
BitArray flags = new BitArray(32);
flags[12] = false;
```

3. Il extrait les bits individuels.

Pour extraire les bits individuels, vous devez les décaler. Par exemple, l'expression suivante apparaît fréquemment, car le décalage à droite de 5 bits équivaut à diviser par 32, car  $2*2*2*2*2 = 2^5 = 32$ . Ainsi, l'expression de décalage suivante localise le **int** contenant le bit à la position **index** :

```
index >> 5
```

4. Il détermine la valeur du bit correct.

Une fois le **int** correct trouvé, le bit individuel (parmi les 32) doit toujours être déterminé. Vous pouvez pour ce faire avoir recours à l'expression suivante :

```
1 << index
```

Pour comprendre comment cela fonctionne, vous devez savoir que lorsque vous décalez un **int** à gauche, seuls les 5 bits de poids faible du second argument sont utilisés. (Seuls 5 bits sont utilisés, car le **int** en cours de décalage compte 32 bits.) En d'autres termes, l'expression de décalage à gauche ci-dessus est sémantiquement identique à la suivante :

```
1 << (index % 32)
```

## Informations complémentaires concernant BitArray

Voici le détail de la classe **BitArray** :

```
class BitArray
{
 public BitArray(int length)
 {
 if (length < 0)
 throw new ArgumentOutOfRangeException(...);
 this.bits = new int[((length - 1) >> 5) + 1];
 this.length = length;
 }

 public int Length
 {
 get { return length; }
 }

 public bool this[int index]
 {
 get {
 BoundsCheck(index);
 return (bits[index >> 5] & (1 << index)) != 0;
 }
 set {
 BoundsCheck(index);
 if (value) {
 bits[index >> 5] |= (1 << index);
 } else {
 bits[index >> 5] &= ~(1 << index);
 }
 }
 }

 private void BoundsCheck(int index)
 {
 if (index < 0 || index >= length) {
 throw new ArgumentOutOfRangeException(...);
 }
 }

 private int[] bits;
 private int length;
}
```

## Atelier 13.1 : Utilisation de propriétés et d'indexeurs



---

### Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- créer des propriétés pour encapsuler des données dans une classe ;
- définir des indexeurs permettant d'accéder à des classes à l'aide d'une notation de type tableau.

### Conditions préalables

Pour pouvoir aborder cet atelier, vous devez disposer des connaissances suivantes :

- création et utilisation de classes ;
- utilisation de tableaux et de collections.

**Durée approximative de cet atelier : 30 minutes**

## Exercice 1

### Amélioration de la classe Account

Dans cet exercice, vous supprimerez les méthodes de numéro de compte bancaire et de type de compte bancaire de la classe **BankAccount** et vous les remplacerez par des propriétés en lecture seule. Vous ajouterez également à la classe **BankAccount** une propriété de chaîne en lecture/écriture destinée à contenir le nom du titulaire du compte.

#### ► Pour changer les méthodes de numéro et de type de compte en propriétés en lecture seule

1. Ouvrez le projet Bank.sln, situé dans le dossier *dossier d'installation\Labs\Lab13\Exercice1\Starter\Bank*.
2. Dans la classe **BankAccount**, remplacez la méthode appelée **Number** par une propriété en lecture seule (propriété dotée d'un accesseur **get**, mais pas d'un accesseur **set**). Ceci est illustré dans le code suivant :

```
public long Number
{
 get { return accNo; }
}
```

3. Compilez le projet.  
Vous obtiendrez des messages d'erreur. La raison en est que **BankAccount.Number** est toujours utilisé en tant que méthode dans les quatre méthodes surchargées **Bank.CreateAccount**.
4. Changez ces quatre méthodes **Bank.CreateAccount** pour qu'elles accèdent au numéro de compte bancaire en tant que propriété.

Par exemple, modifiez :

```
long accNo = newAcc.Number();
```

en :

```
long accNo = newAcc.Number;
```

5. Enregistrez et compilez le projet.
6. Dans la classe **BankAccount**, remplacez la méthode appelée **Type** par une propriété en lecture seule dont l'accesseur **get** retourne **accType.ToString**.
7. Enregistrez et compilez le projet.

► **Pour ajouter à la classe `BankAccount` une propriété en lecture/écriture destinée à contenir le nom du titulaire du compte**

1. Ajoutez un champ privé *holder* de type `string` à la classe **`BankAccount`**.
2. Ajoutez une propriété en lecture/écriture appelée **`Holder`** (notez le « H » majuscule) de type `string` à la classe **`BankAccount`**.

Les accesseurs **`get`** et **`set`** de cette propriété utiliseront la chaîne *holder* que vous venez de créer :

```
public string Holder
{
 get { return holder; }
 set { holder = value; }
}
```

3. Modifiez la méthode **`BankAccount.ToString`** de manière à ce que la chaîne qu'elle retourne contienne le nom du titulaire du compte ainsi que le numéro, le type et le solde du compte.
4. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.

► **Pour tester les propriétés**

1. Ouvrez le test de validation `TestHarness.sln`, situé dans le dossier *dossier d'installation\Labs\Lab13\Exercise1\Starter\TestHarness*.
2. Ajoutez une référence à la bibliothèque `Bank` (DLL contenant les composants sur lesquels vous avez travaillé lors des deux procédures précédentes) en procédant en deux étapes comme suit :
  - a. Développez le projet dans l'Explorateur de solutions.
  - b. Cliquez avec le bouton droit sur **Références**, puis cliquez sur **Ajouter une référence**.
  - c. Cliquez sur **Parcourir**.
  - d. Accédez au dossier *dossier d'installation\Labs\Lab13\Exercise1\Starter\Bank\bin\Debug*.
  - e. Cliquez sur **Bank.dll**, sur **Ouvrir**, puis sur **OK**.
3. Ajoutez deux instructions à la méthode **`Main`** de la classe **`CreateAccount`**, comme suit :
  - Affectez « Sid » au nom du titulaire de *acc1*.
  - Affectez « Ted » au nom du titulaire de *acc2*.
4. Ajoutez les instructions destinées à extraire et afficher le numéro et le type de chaque compte.
5. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.
6. Lancez le projet et vérifiez que les numéros de compte, les types de compte et les noms « Sid » et « Ted » s'affichent.

## Exercice 2

### Modification de la classe Transaction

Dans cet exercice, vous modifierez la classe **BankTransaction** (que vous avez développée au cours d'ateliers précédents et qui est fournie ici). Vous vous souvenez sans doute que la classe **BankTransaction** a été créée pour contenir des informations relatives à une transaction financière appartenant à un objet **BankAccount**.

Vous remplacerez les méthodes **When** et **Amount** par deux propriétés en lecture seule. (**When** retourne la date de la transaction, **Amount**, le montant de la transaction).

#### ► Pour changer la méthode When en propriété en lecture seule

1. Ouvrez le projet Bank.sln, situé dans le dossier *dossier d'installation\Labs\Lab13\Exercice2\Starter\Bank*.
2. Dans la classe **BankTransaction**, remplacez la méthode appelée **When** par une méthode en lecture seule portant le même nom.
3. Compilez le projet.  
Vous obtiendrez un message d'erreur. La raison en est que **BankTransaction.When** est toujours utilisée en tant que méthode dans **Audit.RecordTransaction**. (La classe **Audit** enregistrant un journal d'audit contenant des informations relatives aux transactions, elle utilise les méthodes **When** et **Amount** pour trouver la date et le montant de chaque transaction.)
4. Modifiez la méthode **Audit.RecordTransaction** de manière à ce qu'elle accède au membre **When** en tant que propriété.
5. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.

#### ► Pour changer la méthode Amount en propriété en lecture seule

1. Dans la classe **BankTransaction**, remplacez la méthode appelée **Amount** par une méthode en lecture seule.
2. Compilez le projet.  
Vous obtiendrez des messages d'erreur. La raison en est que **BankTransaction.Amount** est toujours utilisée en tant que méthode dans **Audit.RecordTransaction**.
3. Modifiez la méthode **Audit.RecordTransaction** de manière à ce qu'elle accède au membre **Amount** en tant que propriété.
4. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.

► **Pour tester les propriétés**

1. Ouvrez le test de validation TestHarness.sln, situé dans le dossier *dossier d'installation\Labs\Lab13\Exercise2\Starter\TestHarness*.
2. Ajoutez une référence à la bibliothèque Bank (DLL contenant les composants sur lesquels vous avez travaillé lors des procédures précédentes) en procédant en deux étapes comme suit :
  - a. Développez le projet dans l'Explorateur de solutions.
  - b. Cliquez avec le bouton droit sur **Références**, puis cliquez sur **Ajouter une référence**.
  - c. Cliquez sur **Parcourir**.
  - d. Accédez au dossier *dossier d'installation\Labs\Lab13\Exercise2\Starter\Bank\bin\Debug*.
  - e. Cliquez sur **Bank.dll**, sur **Ouvrir**, puis sur **OK**.
3. Ajoutez des instructions à la méthode **Main** de la classe **CreateAccount**, qui effectueront les actions suivantes :
  - a. Déposer de l'argent sur les comptes *acc1* et *acc2*. (Faites appel à la méthode **Deposit** et créez vos propres numéros).
  - b. Retirer de l'argent des comptes *acc1* et *acc2*. (Faites appel à la méthode **Withdraw**).
  - c. Afficher l'historique des transactions de chaque compte. Une méthode appelée **Write** est fournie à la fin du test de validation. Vous lui transmettez un compte dont vous voulez afficher l'historique des transactions. Elle utilise les propriétés **When** et **Amount** de la classe **BankTransaction** et les teste. Vous trouverez ci-dessous un exemple :  
`Write(acc1);`
4. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.
5. Lancez le projet et vérifiez que les informations concernant les transactions s'affichent correctement.



## Exercice 3

### Création et utilisation d'un indexeur

Dans cet exercice, vous ajouterez un indexeur à la classe **BankAccount** qui fournira un accès aux objets **BankTransaction** placés dans le cache du tableau interne.

Les transactions appartenant à un compte sont accessibles par le biais d'une file d'attente (**System.Collections.Queue**) se trouvant dans l'objet **BankAccount**.

Vous définirez un indexeur dans la classe **BankAccount** qui extrait la transaction à l'endroit spécifié de la file d'attente ou retourne la valeur **null** si aucune transaction n'existe à cet endroit. Par exemple :

```
myAcc.AccountTransactions[2]
```

retourne la transaction numéro 2, qui est la troisième dans la file d'attente.

La méthode **GetEnumerator** de **System.Collections.Queue** sera utile au cours de cet exercice.

#### ► Pour déclarer un indexeur **BankAccount** en lecture seule

1. Ouvrez le projet **Bank.sln**, situé dans le dossier *dossier d'installation\Labs\Lab13\Exercise3\Starter\Bank*.
2. Dans la classe **BankAccount**, déclarez un indexeur public qui retourne un objet **BankTransaction** et prend un seul paramètre **int** appelé **index**, comme suit :

```
public BankTransaction this[int index]
{
 ...
}
```

3. Ajoutez un accesseur **get** au corps de l'indexeur et implémentez-le avec une seule instruction **return new BankTransaction(99);** comme suit :

```
public BankTransaction this[int index]
{
 get { return new BankTransaction(99); }
}
```

L'objectif de cette étape est seulement de tester la syntaxe de l'indexeur. Vous implémenterez l'indexeur correctement plus tard.

4. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.

► **Pour créer des transactions**

1. Ouvrez le test de validation TestHarness.sln, situé dans le dossier *dossier d'installation\Labs\Lab13\Exercise3\Starter\TestHarness*.
2. Ajoutez une référence à la bibliothèque Bank (DLL contenant les composants sur lesquels vous avez travaillé au cours de l'étape précédente) en procédant en deux étapes comme suit :
  - a. Développez le projet dans l'Explorateur de solutions.
  - b. Cliquez avec le bouton droit sur **Références**, puis cliquez sur **Ajouter une référence**.
  - c. Cliquez sur **Parcourir**.
  - d. Accédez au dossier *dossier d'installation\Labs\Lab13\Exercise3\Starter\Bank\bin\Debug*.
  - e. Cliquez sur **Bank.dll**, sur **Ouvrir**, puis sur **OK**.
3. Créez quelques transactions en ajoutant les instructions suivantes à la fin de la méthode **CreateAccount.Main** :

```
for (int i = 0; i < 5; i++) {
 acc1.Deposit(100);
 acc1.Withdraw(50);
}
Write(acc1);
```

Les appels à **Deposit** et **Withdraw** créent des transactions.
4. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.

Lancez le projet et vérifiez que les transactions **Deposit** et **Withdraw** s'affichent correctement.

### ► Pour appeler l'indexeur de **BankAccount**

1. Les dernières instructions de la méthode **CreateAccount.Write** affichent actuellement les transactions à l'aide d'une instruction **foreach** comme suit :

```
Queue tranQueue = acc.Transactions();
foreach (BankTransaction tran in tranQueue) {
 Console.WriteLine("Date : {0}\tMontant : {1}",
 ↪tran.When, tran.Amount);
}
```

2. Modifiez l'affichage des transactions comme suit :

- a. Remplacez cette instruction **foreach** par une instruction **for** incrémentant une variable **int** appelée *counter* de zéro à la valeur retournée par **tranQueue.Count**.
- b. Dans l'instruction **for**, appelez l'indexeur **BankAccount** que vous avez déclaré dans la procédure précédente. Utilisez *counter* comme paramètre d'indice et enregistrez le **BankTransaction** retourné dans une variable locale appelée *tran*.

- c. Affichez les informations contenues dans *tran* :

```
for (int counter = 0; counter < tranQueue.Count;
 ↪counter++) {
 BankTransaction tran = acc[counter];
 Console.WriteLine("Date : {0}\tMontant : {1}",
 ↪tran.When, tran.Amount);
}
```

3. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.
4. Exécutez le projet.

Il affiche une série de transactions dont la valeur est 99 (valeur de test provisoire que vous avez utilisée précédemment), car l'indexeur n'a pas été implémenté dans son intégralité.

► **Pour terminer l'implémentation de l'indexeur BankAccount**

1. Retournez au projet Bank.sln (dans le dossier *dossier d'installation\Labs\Lab13\Exercice3\Starter\Bank*).
2. Dans la classe **BankAccount**, supprimez l'instruction `return new BankTransaction(99);` du corps de l'indexeur.
3. Les transactions **BankAccount** se trouvent dans un champ privé appelé *tranQueue* de type **System.Collections.Queue**. Cette classe **Queue** ne possédant pas d'indexeur, vous avez besoin pour accéder à un élément donné de parcourir manuellement la classe à l'aide d'une boucle. La procédure est la suivante :
  - a. Déclarez une variable de type *IEnumerator* et initialisez-la à l'aide de la méthode **GetEnumerator** de *tranQueue*. (Toutes les files d'attente fournissent un énumérateur vous permettant de les parcourir pas à pas).
  - b. Effectuez *n* itérations dans la file d'attente en utilisant la méthode **MoveNext** de la variable *IEnumerator* pour vous déplacer vers le prochain élément de la file d'attente.
  - c. Retournez la **BankTransaction** trouvée au *nième* emplacement.

Votre code doit se présenter comme suit :

```
IEnumerator ie = tranQueue.GetEnumerator();
for (int i = 0; i <= index; i++) {
 ie.MoveNext();
}
BankTransaction tran = (BankTransaction)ie.Current;
return tran;
```

4. Vérifiez que le paramètre **int index** n'est ni supérieur à **tranQueue.Count**, ni inférieur à zéro.

Vérifiez-le en parcourant **tranQueue** à l'aide d'une boucle.

5. Voici le code complet de l'indexeur :

```
public BankTransaction this[int index]
{
 get
 {
 if (index < 0 || index >= tranQueue.Count)
 return null;

 IEnumerator ie = tranQueue.GetEnumerator();
 for (int i = 0; i <= index; i++) {
 ie.MoveNext();
 }
 BankTransaction tran = (BankTransaction)ie.Current;
 return tran;
 }
}
```

6. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.
7. Retournez à TestHarness et lancez-le.  
Vérifiez que les dix transactions s'affichent correctement.

## Contrôle des acquis

- Utilisation des propriétés
- Utilisation des indexeurs

1. Déclarez une classe **Font** contenant une propriété en lecture seule appelée **Name** de type **string**.
2. Déclarez une classe **DialogBox** contenant une propriété en lecture/écriture appelée **Caption** de type **string**.

3. Déclarez une classe **MutableString** (pour chaînes modifiables) contenant un indexeur en lecture/écriture de type **char** attendant un seul paramètre **int**.
  
4. Déclarez une classe **Graph** contenant un indexeur en lecture seule de type **double** attendant un seul paramètre de type **Point**.

---

## Module 14 : Attributs

### Table des matières

|                                                      |    |
|------------------------------------------------------|----|
| Vue d'ensemble                                       | 1  |
| Vue d'ensemble des attributs                         | 2  |
| Définition d'attributs personnalisés                 | 13 |
| Extraction de valeurs d'attribut                     | 22 |
| Atelier 14.1 : Définition et utilisation d'attributs | 26 |
| Contrôle des acquis                                  | 34 |



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.



# Vue d'ensemble

- Vue d'ensemble des attributs
- Définition d'attributs personnalisés
- Extraction de valeurs d'attribut

Les attributs constituent une technique simple pour ajouter des métadonnées à des classes. Ils peuvent être utiles pour générer des composants.

Dans ce module, vous allez découvrir le rôle des attributs et la fonction qu'ils assurent dans les applications C#. Vous allez apprendre la syntaxe propre aux attributs et découvrir comment utiliser certains attributs prédéfinis dans l'environnement Microsoft® .NET Framework. Vous apprendrez également à créer des attributs personnalisés définis par l'utilisateur. Enfin, vous découvrirez comment les classes et autres types d'objets peuvent implémenter et utiliser ces attributs personnalisés pour obtenir des informations d'attribut au moment de l'exécution.

À la fin de ce module, vous serez à même d'effectuer les tâches suivantes :

- utiliser des attributs prédéfinis courants ;
- créer des attributs personnalisés simples ;
- lancer des requêtes pour obtenir des informations sur les attributs au moment de l'exécution.

## ◆ Vue d'ensemble des attributs

- **Présentation des attributs**
- **Application d'attributs**
- **Attributs prédéfinis courants**
- **Utilisation de l'attribut Conditional**
- **Utilisation de l'attribut DllImport**
- **Utilisation de l'attribut Transaction**

---

En introduisant les attributs, le langage C# offre une technique pratique pour réaliser des tâches telles que changer le comportement du runtime, obtenir des informations de transaction à propos d'un objet, transmettre des informations d'organisation à un concepteur et manipuler du code non managé.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- identifier les tâches que vous pouvez réaliser avec des attributs ;
- employer la syntaxe pour utiliser des attributs dans votre code ;
- identifier certains attributs prédéfinis disponibles dans le .NET Framework.

## Présentation des attributs

### ■ Les attributs sont :

- des balises déclaratives qui transmettent des informations au runtime ;
- stockés avec les métadonnées de l'élément associé.

### ■ Le .NET Framework offre des attributs prédéfinis

- Le runtime contient du code permettant d'examiner les valeurs des attributs et d'agir sur elles

Le .NET Framework fournit des attributs pour vous permettre d'étendre les capacités du langage C#. Un attribut est une balise déclarative que vous utilisez pour transmettre au runtime des informations sur le comportement d'éléments de programmation tels que les classes, les énumérateurs et les assemblés.

Vous pouvez vous représenter les attributs comme des annotations que vos programmes peuvent stocker et utiliser. Dans la plupart des cas, vous écrivez le code qui extrait les valeurs d'un attribut en plus du code qui vient modifier le comportement au moment de l'exécution. Dans sa forme la plus simple, un attribut est un autre moyen de documenter votre code.

Vous pouvez appliquer des attributs à de nombreux éléments du code source. Les informations sur les attributs sont stockées conjointement aux métadonnées des éléments auxquels ils sont associés.

Le .NET Framework est doté de plusieurs attributs prédéfinis. Le code permettant de les examiner et d'agir sur les valeurs qu'ils contiennent est également incorporé en tant qu'élément du runtime et du Kit de développement Microsoft .NET Framework SDK.

## Application d'attributs

### ■ Syntaxe : utilisez des crochets pour spécifier un attribut

```
[attribut(paramètres_positionnels,paramètre_nommé=valeur, ...)]
élément
```

### ■ Pour appliquer plusieurs attributs à un élément, vous pouvez :

- spécifier plusieurs attributs en les séparant par des crochets ;
- utiliser une seule paire de crochets en séparant chaque attribut par une virgule ;
- pour certains éléments, tels que les assembls, spécifier explicitement le nom de l'élément associé à l'attribut.

---

Vous pouvez appliquer des attributs à différents types d'éléments de programmation, tels que les assembls, les modules, les classes, les structs, les enums, les constructeurs, les méthodes, les propriétés, les champs, les événements, les interfaces, les paramètres, les valeurs de retour et les délégués.

## Syntaxe d'un attribut

Pour spécifier un attribut et l'associer à un élément de programmation, utilisez la syntaxe générale suivante :

```
[attribut(paramètres_positionnels,paramètre_nommé=valeur, ...)]
élément
```

Vous spécifiez un nom d'attribut et ses valeurs entre crochets ([ et ]) avant l'élément de programmation auquel vous voulez l'appliquer. La plupart des attributs prennent un ou deux paramètres, qui peuvent être *positionnels* (*positional*) ou *nommés* (*named*).

Vous spécifiez un paramètre positionnel à une position définie de la liste de paramètres, comme pour spécifier les paramètres d'une méthode. Toute valeur de paramètre nommé est placée à la suite des paramètres positionnels. Les paramètres positionnels permettent de spécifier des informations essentielles, tandis que les paramètres nommés servent à véhiculer des informations facultatives dans un attribut.

---

**Conseil** Avant d'utiliser un attribut qui ne vous est pas familier, il est conseillé de consulter la documentation relative à cet attribut afin de savoir quels paramètres sont disponibles et s'ils doivent être positionnels ou nommés.

---

## Exemple

Comme exemple d'utilisation des attributs, examinez le code suivant, dans lequel l'attribut **DefaultEvent** est appliqué sur une classe à l'aide d'un paramètre **string** positionnel, **ShowResult** :

```
using System.ComponentModel;
...
[DefaultEvent("ShowResult")]
public class Calculator: System.Windows.Forms.UserControl
{
 ...
}
```

## Application d'attributs multiples

Vous pouvez appliquer plusieurs attributs à un élément. Vous pouvez placer chaque attribut entre crochets, ou bien en placer plusieurs entre crochets, mais séparés par des virgules.

Dans certaines circonstances, vous devez spécifier exactement à quel élément est associé un attribut. Par exemple, dans le cas d'attributs d'assembly, placez-les après toute clause **using** mais avant le code, et spécifiez-les explicitement en tant qu'attributs de l'assembly.

L'exemple suivant montre comment utiliser l'attribut d'assembly **CLSCompliant**. Cet attribut indique si un assembly est en tous points conforme à CLS (Common Language Specification) ou non.

```
using System;
[assembly:CLSCompliant(true)]

class MyClass
{
 ...
}
```

## Attributs prédéfinis courants

### ■ .NET offre de nombreux attributs prédéfinis

- Attributs généraux
- Attributs d'interopérabilité COM
- Attributs de gestion des transactions
- Attributs de génération de composants destinés à un concepteur visuel

---

Les capacités des attributs prédéfinis du .NET Framework couvrent un large éventail de domaines, de l'interopérabilité avec COM à la compatibilité avec les outils de conception visuelle.

Cette rubrique décrit certains attributs prédéfinis courants fournis dans le .NET Framework. Elle ne prétend cependant pas être exhaustive. Pour plus d'informations sur les attributs prédéfinis, consultez l'aide de Microsoft Visual Studio® .NET.

### Attributs généraux

Le tableau ci-dessous présente quelques attributs généraux fournis par le .NET Framework.

| Attribut           | S'applique à | Description                                                                                                                                                                                                                                                      |
|--------------------|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Conditional</b> | Méthode      | Teste si un symbole nommé est défini. Si tel est le cas, tous les appels à la méthode sont exécutés normalement. Dans le cas contraire, l'appel n'est pas généré.                                                                                                |
| <b>DllImport</b>   | Méthode      | Indique que la méthode est implémentée dans du code non managé, dans la bibliothèque de liaisons dynamiques (DLL, <i>dynamic link library</i> ) spécifiée. Provoque le chargement de la DLL au moment de l'exécution ainsi que l'exécution de la méthode nommée. |

## Attributs d'interopérabilité COM

Lorsque vous utilisez les attributs pour assurer une interopérabilité avec COM, vous devez veiller à ce que l'utilisation de composants COM à partir de l'environnement managé du .NET Framework soit aussi transparente que possible. Le .NET Framework possède de nombreux attributs liés à l'interopérabilité COM. Certains d'entre eux sont présentés dans le tableau ci-dessous.

| Attribut                     | S'applique à                        | Description                                                                                                                             |
|------------------------------|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| <b>ComImport</b>             | Classe/Interface                    | Indique qu'une définition de classe ou d'interface a été importée d'une bibliothèque de types COM.                                      |
| <b>ComRegisterFunction</b>   | Méthode                             | Spécifie la méthode à appeler lorsqu'un assembly .NET Framework est inscrit dans le but de l'utiliser à partir de COM.                  |
| <b>ComUnregisterFunction</b> | Méthode                             | Spécifie la méthode à appeler lorsque l'inscription d'un assembly .NET Framework dans le but de l'utiliser à partir de COM est annulée. |
| <b>DispId</b>                | Méthode, champ, propriété           | Indique quel ID de dispatch doit être utilisé pour la méthode, le champ ou la propriété.                                                |
| <b>In</b>                    | Paramètre                           | Indique que les données doivent être marshalées de l'appelant vers l'appelé.                                                            |
| <b>MarshalAs</b>             | Champ, paramètre, valeurs de retour | Spécifie comment les données doivent être marshalées entre COM et l'environnement managé.                                               |
| <b>ProgId</b>                | Classe                              | Spécifie quel identificateur programmatique (Prog ID) doit être utilisé pour la classe.                                                 |
| <b>Out</b>                   | Paramètre                           | Indique que les données doivent être marshalées de l'appelé vers l'appelant.                                                            |
| <b>InterfaceType</b>         | Interface                           | Spécifie si une interface managée est <b>IDispatch</b> , <b>IUnknown</b> ou « dual » lorsqu'elle est exposée à COM.                     |

Pour plus d'informations sur l'interopérabilité COM, recherchez « Microsoft ComServices » dans les documents de l'aide du Kit de développement Microsoft .NET Framework SDK.

## Attributs de gestion des transactions

Les composants qui s'exécutent dans un environnement COM+ utilisent la gestion des transactions. L'attribut que vous utilisez à cet effet est indiqué dans le tableau ci-dessous.

| Attribut           | S'applique à | Description                                                              |
|--------------------|--------------|--------------------------------------------------------------------------|
| <b>Transaction</b> | Classe       | Spécifie le type de transaction qui doit être disponible pour cet objet. |

## Attributs de génération de composants destinés à un concepteur visuel

Les développeurs qui génèrent des composants pour un concepteur visuel utilisent les attributs répertoriés dans le tableau ci-dessous.

| Attribut               | S'applique à         | Description                                                                                                                                                                                                                                                    |
|------------------------|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Bindable</b>        | Propriété            | Spécifie si une propriété est généralement utilisée pour la liaison.                                                                                                                                                                                           |
| <b>DefaultProperty</b> | Classe               | Spécifie la propriété par défaut du composant.                                                                                                                                                                                                                 |
| <b>DefaultValue</b>    | Propriété            | Indique que la propriété est la valeur par défaut du composant.                                                                                                                                                                                                |
| <b>Localizable</b>     | Propriété            | Lorsque du code est généré pour un composant, les valeurs des propriétés des membres qui portent la marque <b>Localizable(true)</b> sont enregistrées dans les fichiers de ressources. Vous pouvez localiser ces fichiers de ressources sans modifier le code. |
| <b>DefaultEvent</b>    | Classe               | Spécifie l'événement par défaut du composant.                                                                                                                                                                                                                  |
| <b>Catégorie</b>       | Propriété, événement | Spécifie la catégorie dans laquelle le concepteur visuel doit placer cette propriété ou cet événement dans la fenêtre Propriétés.                                                                                                                              |
| <b>Description</b>     | Propriété, événement | Définit un texte bref à afficher au bas de la fenêtre Propriétés dans le concepteur visuel lorsque cette propriété ou cet événement est sélectionné.                                                                                                           |



## Utilisation de l'attribut Conditional

### ■ Sert d'outil de débogage

- Provoque la compilation conditionnelle des appels de méthode, selon la valeur d'un symbole défini par le programmeur
- Ne provoque pas la compilation conditionnelle de la méthode elle-même

### ■ Restrictions des méthodes

- Doivent avoir un type de retour **void**
- Ne doivent pas être déclarées comme **override**
- Ne doivent pas provenir d'une interface héritée

```
using System.Diagnostics;
...
class MyClass
{
 [Conditional ("DEBUGGING")]
 public static void MyMethod()
 {
 ...
 }
}
```

Vous pouvez utiliser l'attribut **Conditional** comme une aide au débogage dans votre code C#. Cet attribut provoque la compilation conditionnelle des appels de méthode, selon la valeur d'un symbole que vous définissez. Il vous permet d'appeler des méthodes qui, par exemple, affichent les valeurs des variables tandis que vous testez et déboguez le code. Après avoir débogué votre programme, vous pouvez « annuler la définition » du symbole et recompiler votre code sans rien changer d'autre. (Vous pouvez aussi simplement supprimer le symbole depuis la ligne de commande, sans rien changer.)

## Exemple

L'exemple ci-dessous montre comment utiliser l'attribut **Conditional**. Dans cet exemple, la méthode **MyMethod** de **MyClass** est balisée à l'aide de l'attribut **Conditional** par le symbole **DEBUGGING** :

```
using System.Diagnostics;
...
class MyClass
{
 [Conditional ("DEBUGGING")]
 public static void MyMethod()
 {
 ...
 }
}
```

Le symbole `DEBUGGING` est défini comme suit :

```
#define DEBUGGING

class AnotherClass
{
 public static void Test()
 {
 MyClass.MyMethod();
 }
}
```

Tant que le symbole `DEBUGGING` reste défini lorsque l'appel de méthode est compilé, l'appel de méthode fonctionne normalement. Lorsque `DEBUGGING` n'est pas défini, le compilateur omet les appels à la méthode. Par conséquent, lorsque vous exécutez le programme, il est traité comme si cette ligne de code n'existait pas.

Vous pouvez définir le symbole de deux manières : soit en ajoutant une directive **`#define`** au code, comme dans l'exemple précédent, soit en définissant le symbole à partir de la ligne de commande lorsque vous compilez votre programme.

## Restrictions des méthodes

Les méthodes auxquelles vous pouvez appliquer un attribut **Conditional** sont sujettes à certaines restrictions. Elles doivent notamment disposer d'un type de retour **void**, ne pas être marquées comme **override** et ne pas être l'implémentation d'une méthode d'interface héritée.

---

**Remarque** L'attribut **Conditional** ne provoque pas la compilation conditionnelle de la méthode elle-même. Il détermine uniquement l'action qui se produit lorsque la méthode est appelée. Si vous voulez la compilation conditionnelle d'une méthode, vous devez utiliser les directives **`#if`** et **`#endif`** dans votre code.

---

## Utilisation de l'attribut DllImport

### ■ Avec l'attribut DllImport, vous pouvez :

- appeler du code non managé dans des DLL à partir d'un environnement C# ;
- baliser une méthode externe pour indiquer qu'elle réside dans une DLL non managée.

```
using System.Runtime.InteropServices;
...
public class MyClass()
{
 [DllImport("MyDLL.dll", EntryPoint="MyFunction")]
 public static extern int MyFunction(string param1);
 ...
 int result = MyFunction("Bonjour code non managé");
 ...
}
```

Vous pouvez utiliser l'attribut **DllImport** pour appeler du code non managé dans vos programmes C#. Le *code non managé* désigne le code qui a été développé hors de l'environnement .NET (c'est-à-dire, du code C standard compilé dans des fichiers DLL). En utilisant l'attribut **DllImport**, vous pouvez appeler du code non managé résidant dans des bibliothèques de liaisons dynamiques (DLL) à partir de votre environnement C# managé.

### Appel de code non managé

L'attribut **DllImport** vous permet de baliser une méthode **extern** comme résidant dans une DLL non managée. Lorsque votre code appelle cette méthode, le Common Language Runtime repère la DLL, la charge dans la mémoire de votre processus, marshale les paramètres nécessaires et transfère le contrôle à l'adresse de début du code non managé. Cette manière de procéder est totalement différente des programmes normaux, qui n'ont pas d'accès direct à la mémoire qui leur est allouée. Le code suivant montre un exemple d'appel de code non managé :

```
using System.Runtime.InteropServices;
...
public class MyClass()
{
 [DllImport("MyDLL.dll", EntryPoint="MyFunction")]
 public static extern int MyFunction(string param1);
 ...
 int result = MyFunction("Bonjour code non managé");
 ...
}
```

## Utilisation de l'attribut Transaction

### ■ Pour gérer des transactions dans COM+

- Spécifiez que votre composant doit être inclus en cas de demande de validation
- Utilisez un attribut Transaction sur la classe qui implémente le composant

```
using System.EnterpriseServices;
...
[Transaction(TransactionOption.Required)]
public class MyTransactionalComponent
{
 ...
}
```

---

En tant que développeur Microsoft Visual Basic® ou C++ travaillant dans un environnement Microsoft, vous êtes probablement familiarisé avec les technologies telles que COM+. COM+ possède une fonctionnalité importante qui vous permet de développer des composants pouvant participer à des transactions distribuées, c'est-à-dire des transactions capables de couvrir plusieurs bases de données, ordinateurs ou composants.

### Gestion des transactions dans COM+

Écrire du code pour garantir une validation de transaction correcte dans un environnement distribué est difficile. Toutefois, si vous utilisez COM+, il prend soin de gérer l'intégrité transactionnelle du système et de coordonner les événements sur le réseau.

Dans ce cas, vous devez juste spécifier que votre composant doit être inclus lorsqu'une application l'utilisant demande une validation de transaction. Pour ce faire, vous pouvez utiliser un attribut **Transaction** sur la classe qui implémente le composant, comme suit :

```
using System.EnterpriseServices;
...
[Transaction(TransactionOption.Required)]
public class MyTransactionalComponent
{
 ...
}
```

L'attribut **Transaction** est l'un des attributs .NET Framework prédéfinis que le runtime du .NET Framework interprète automatiquement.

## ◆ Définition d'attributs personnalisés

- Définition de la portée d'un attribut personnalisé
- Définition d'une classe d'attribut
- Traitement des attributs personnalisés
- Utilisation d'attributs multiples

Dans le cas où aucun des attributs prédéfinis du .NET Framework ne répond à vos besoins, vous pouvez créer votre propre attribut. Cet attribut personnalisé fournit des propriétés qui vous permettent de stocker et d'extraire des informations à partir de l'attribut.

Comme les attributs prédéfinis, les attributs personnalisés sont des objets qui sont associés à un ou plusieurs éléments de programmation. Ils sont stockés avec les métadonnées des éléments qui leurs sont associés et fournissent aux programmes des mécanismes permettant d'extraire leurs valeurs.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- définir vos propres attributs personnalisés ;
- utiliser vos propres attributs personnalisés.

## Définition de la portée d'un attribut personnalisé

### ■ Utilisez la balise **AttributeUsage** pour définir la portée

#### ● Exemple

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute: System.Attribute
{ ... }
```

### ■ Utilisez l'opérateur de bits « ou » (|) pour spécifier plusieurs éléments

#### ● Exemple

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttribute: System.Attribute
{ ... }
```

Comme avec certains attributs prédéfinis, vous devez explicitement spécifier l'élément de programmation auquel vous voulez appliquer un attribut personnalisé. Pour ce faire, vous annotez votre attribut personnalisé avec une balise **AttributeUsage** comme dans l'exemple suivant :

```
[AttributeUsage(éléments_cible)]
public class MyAttribute: System.Attribute
{ ... }
```

## Définition de la portée d'un attribut

Le paramètre de **AttributeUsage** contient des valeurs provenant de l'énumération **System.AttributeTargets** pour spécifier de quelle manière l'attribut personnalisé peut être utilisé. Les membres de cette énumération sont résumés dans le tableau suivant.

| Membre      | L'attribut peut être appliqué à l'élément suivant |
|-------------|---------------------------------------------------|
| Class       | Classe                                            |
| Constructor | Constructeur                                      |
| Delegate    | Délégué                                           |
| Enum        | Enum                                              |
| Event       | Événement                                         |
| Field       | Champ                                             |
| Interface   | Interface                                         |
| Method      | Méthode                                           |
| Module      | Module                                            |

*(suite)*

| Membre      | L'attribut peut être appliqué à l'élément suivant |
|-------------|---------------------------------------------------|
| Parameter   | Paramètre                                         |
| Property    | Propriété                                         |
| ReturnValue | Valeur de retour                                  |
| Struct      | Struct                                            |
| Assembly    | Assembly                                          |
| All         | N'importe quel élément                            |

## Exemple d'utilisation d'attributs personnalisés

Pour spécifier que l'attribut personnalisé **MyAttribute** peut être appliqué uniquement aux méthodes, utilisez le code suivant :

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute: System.Attribute
{
 ...
}
```

## Spécification d'éléments multiples

Si l'attribut peut être appliqué à plusieurs types d'éléments, utilisez l'opérateur de bits « ou » (|) pour spécifier plusieurs types de cibles. Par exemple, si **MyAttribute** peut être également appliqué aux constructeurs, le code précédent sera modifié comme suit :

```
[AttributeUsage(AttributeTargets.Method |
 ↪AttributeTargets.Constructor)]
public class MyAttribute: System.Attribute
{
 ...
}
```

Si un développeur essaie d'utiliser **MyAttribute** dans un contexte différent de celui qui est défini par **AttributeUsage**, son code ne pourra pas être compilé.

## Définition d'une classe d'attribut

### ■ Dérivation d'une classe d'attribut

- Toutes les classes d'attributs doivent dériver directement ou indirectement de la classe `System.Attribute`
- Ajoutez le suffixe « `Attribute` » au nom d'une classe d'attribut

### ■ Composants d'une classe d'attribut

- Définissez un seul constructeur pour chaque classe d'attribut à l'aide d'un paramètre positionnel
- Utilisez les propriétés pour définir une valeur facultative à l'aide d'un paramètre nommé

---

Après avoir défini la portée d'un attribut personnalisé, vous devez spécifier son comportement. Pour ce faire, vous devez définir une classe d'attribut. Cette classe définit le nom de l'attribut, son mode de création et les informations qu'il stocke.

Le Kit de développement Microsoft .NET Framework SDK fournit une classe de base, **`System.Attribute`**, que vous devez utiliser pour dériver les classes d'attributs personnalisés et accéder aux valeurs contenues dans les attributs personnalisés.

## Dérivation d'une classe d'attribut

Toutes les classes d'attributs personnalisés doivent dériver directement ou indirectement de la classe **`System.Attribute`**. Le code suivant fournit un exemple :

```
public class DeveloperInfoAttribute: System.Attribute
{
 ...
 public DeveloperInfoAttribute(string developer)
 {
 ...
 }
 public string Date
 {
 get { ... }
 set { ... }
 }
}
```

Il est recommandé d'ajouter le suffixe « `Attribute` » au nom d'une classe d'attribut personnalisé, comme dans **`DeveloperInfoAttribute`**. Cela permet de mieux distinguer les classes d'attributs des autres.



## Composants d'une classe d'attribut

Toutes les classes d'attributs doivent avoir un constructeur. Par exemple, si l'attribut **DeveloperInfo** attend le nom du développeur comme paramètre de type string, il doit disposer d'un constructeur qui accepte un paramètre string.

Un attribut personnalisé doit définir un seul constructeur qui définit les informations obligatoires. Le ou les paramètres positionnels de l'attribut passent ces informations au constructeur. Lorsqu'un attribut possède des données facultatives, il essaie de surcharger le constructeur. Ce n'est pas une bonne technique à adopter. Utilisez les paramètres nommés pour fournir des données facultatives.

Une classe d'attribut peut, toutefois, fournir des propriétés pour obtenir et définir des données. Vous devez par conséquent utiliser des propriétés pour définir les valeurs facultatives, si nécessaire. Un développeur peut ensuite spécifier les valeurs facultatives comme paramètres nommés lorsqu'il utilise l'attribut.

Par exemple, la classe **DeveloperInfoAttribute** fournit une propriété **Date**. Vous pouvez appeler la méthode **set** de la propriété **Date** pour définir le paramètre nommé : *Date*. Le nom du développeur, *Bertrand*, par exemple, est le paramètre positionnel qui est passé au constructeur de l'attribut :

```
[DeveloperInfoAttribute("Bertrand", Date="08-11-2002")]
public class MyClass
{
 ...
}
```

## Traitement des attributs personnalisés

**Le processus de compilation :**

- 1. Recherche la classe de l'attribut**
- 2. Vérifie la portée de l'attribut**
- 3. Vérifie la présence d'un constructeur dans l'attribut**
- 4. Crée une instance de l'objet**
- 5. Recherche un paramètre nommé**
- 6. Définit le champ ou la propriété en fonction d'une valeur de paramètre nommé**
- 7. Enregistre l'état actuel de la classe d'attribut**

---

Lorsque le compilateur rencontre un attribut sur un élément de programmation, il suit la procédure suivante pour déterminer comment appliquer l'attribut :

1. Recherche la classe de l'attribut
2. Vérifie la portée de l'attribut
3. Vérifie la présence d'un constructeur dans l'attribut
4. Crée une instance de l'objet
5. Recherche un paramètre nommé
6. Définit le champ ou la propriété en fonction d'une valeur de paramètre nommé
7. Enregistre l'état actuel de la classe d'attribut

Pour être totalement précis, le compilateur vérifie en fait s'il *peut* appliquer l'attribut, puis stocke l'information pour le faire dans les métadonnées. Le compilateur ne crée pas d'instance d'attribut au moment de la compilation.

## Exemple

Pour en savoir plus sur le traitement des attributs par le compilateur, examinez l'exemple suivant :

```
[AttributeUsage(AttributeTargets.Class)]
public class DeveloperInfoAttribute: System.Attribute
{
 ...
}
.....
{

}

[DeveloperInfo("Bertrand", Date="08-11-2002")]
public class MyClass
{
 ...
}
```

---

**Remarque** Comme indiqué dans la rubrique précédente, il est recommandé d'ajouter le suffixe « Attribute » aux noms des classes d'attributs. Cela n'est pas une nécessité à proprement parler. Votre code pourra être compilé normalement même si vous omettez d'ajouter le suffixe indiqué dans l'exemple. Toutefois, l'absence du suffixe Attribute peut engendrer des problèmes lorsque le compilateur recherche les classes. Utilisez donc toujours le suffixe Attribute.

---

## Processus de compilation

Dans l'exemple précédent, lorsque **MyClass** est compilée, le compilateur recherche une classe d'attribut appelée **DeveloperInfoAttribute**. S'il ne la trouve pas, il cherche ensuite **DeveloperInfo**.

Quand il a trouvé **DeveloperInfo**, le compilateur vérifie si l'attribut est autorisé sur une classe. Il recherche ensuite un constructeur qui correspond aux paramètres spécifiés dans l'utilisation de l'attribut. S'il en trouve un, il crée une instance de l'objet en appelant le constructeur à l'aide des valeurs spécifiées.

S'il trouve un paramètre nommé, le compilateur fait correspondre le nom du paramètre avec un champ ou une propriété de la classe d'attribut, puis définit le champ ou la propriété en fonction de la valeur spécifiée. Ensuite, l'état actuel de la classe d'attribut est enregistré dans les métadonnées de l'élément de programme sur lequel il est appliqué.

## Utilisation d'attributs multiples

- Un élément peut avoir plusieurs attributs
  - Définissez chaque attribut séparément
  
- Un élément peut avoir plusieurs fois le même attribut
  - Use AllowMultiple = true

---

Vous pouvez appliquer plusieurs attributs à un élément de programmation et utiliser plusieurs instances du même attribut dans une application.

### Utilisation d'attributs multiples

Vous pouvez appliquer plusieurs attributs à un élément de programmation. Par exemple, le code suivant vous montre comment baliser la classe **FinancialComponent** avec deux attributs : **Transaction** et **DefaultProperty**.

```
[Transaction(TransactionOption.Required)]
[DefaultProperty("Balance")]
public class FinancialComponent: System.Attribute
{
 ...
 public long Balance
 {
 ...
 }
}
```

## Utilisation répétée du même attribut

Le comportement par défaut d'un attribut personnalisé n'autorise pas plusieurs instances de l'attribut. Toutefois, dans certaines circonstances, il peut être judicieux d'autoriser l'utilisation répétée d'un attribut sur le même élément.

L'attribut personnalisé **DeveloperInfo** illustre parfaitement ce propos. Cet attribut vous permet d'enregistrer le nom du développeur qui a écrit une classe. Si plusieurs développeurs ont participé au développement, vous devez utiliser plusieurs fois l'attribut **DeveloperInfo**. Pour qu'un attribut autorise cela, vous devez le marquer comme **AllowMultiple** dans l'attribut **AttributeUsage**, comme suit :

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
public class DeveloperInfoAttribute: System.Attribute
{
 ...
}
```

## ◆ Extraction de valeurs d'attribut

- Examen des métadonnées de classe
- Recherche d'informations sur les attributs

---

Après avoir appliqué des attributs aux éléments de programmation dans votre code, il est utile d'être en mesure de déterminer les valeurs des attributs. Dans cette section, vous allez apprendre à utiliser la réflexion pour examiner les métadonnées d'attribut d'une classe et rechercher dans les classes des informations d'attribut.

À la fin de cette leçon, vous serez à même d'effectuer les tâches suivantes :

- utiliser la réflexion pour examiner les métadonnées d'attribut d'une classe ;
- rechercher dans les classes des informations d'attribut.

## Examen des métadonnées de classe

### ■ Pour rechercher des informations de métadonnées sur une classe :

- Utilisez la classe `MemberInfo` dans `System.Reflection`
- Remplissez un objet `MemberInfo` à l'aide de `System.Type`
- Créez un objet `System.Type` à l'aide de l'opérateur `typeof`

### ■ Exemple

```
System.Reflection.MemberInfo typeInfo;
typeInfo = typeof(MyClass);
```

Le runtime du .NET Framework fournit un mécanisme appelé *réflexion* qui vous permet d'interroger les informations contenues dans les métadonnées. Les informations d'attribut sont stockées dans les métadonnées.

## Utilisation de la classe `MemberInfo`

Le .NET Framework fournit un espace de noms nommé **System.Reflection**, qui contient des classes que vous pouvez utiliser pour examiner les métadonnées. Une classe particulière de cet espace de noms, la classe **MemberInfo**, est très utile si vous avez besoin d'en savoir plus sur les attributs d'une classe.

Pour remplir un tableau **MemberInfo**, vous pouvez utiliser la méthode **GetMembers** de l'objet **System.Type**. Pour créer cet objet, vous utilisez l'opérateur **typeof** avec une classe ou n'importe quel autre élément, comme indiqué dans le code suivant :

```
System.Reflection.MemberInfo[] memberInfoArray;
memberInfoArray = typeof(MyClass).GetMembers();
...
```

Une fois la variable *typeInfo* créée, on peut y rechercher des informations de métadonnées sur la classe **MyClass**.

---

**Conseil** Pour obtenir des informations plus détaillées, par exemple pour découvrir les valeurs des attributs que possède une méthode, vous pouvez utiliser un objet **MethodInfo**. Il existe d'autres classes « Info » :

**ConstructorInfo**, **EventInfo**, **FieldInfo**, **ParameterInfo** et **PropertyInfo**. Les informations détaillées sur l'utilisation de ces classes dépassent la portée de ce cours, mais pour en apprendre plus, vous pouvez rechercher « System.Reflection (espace de noms) » dans la documentation du Kit de développement Microsoft .NET Framework SDK.

---

---

**Remarque** **MemberInfo** est en fait la classe de base abstraite des autres types « Info ».

---

## Recherche d'informations sur les attributs

- **Pour extraire des informations sur les attributs :**

- Utilisez **GetCustomAttributes** pour extraire les informations sur tous les attributs sous forme de tableau

```
System.Reflection.MemberInfo typeInfo;
typeInfo = typeof(MyClass);
object [] attrs = typeInfo.GetCustomAttributes(false);
```

- Itérez sur le tableau et examinez les valeurs de chaque élément du tableau
- Utilisez la méthode **IsDefined** pour savoir si un attribut particulier a été défini pour une classe

---

Une fois que vous avez créé la variable *typeInfo*, vous pouvez y rechercher des informations sur les attributs appliqués à la classe qui lui est associée.

### Extraction d'informations d'attribut

L'objet **MemberInfo** possède une méthode appelée **GetCustomAttributes**. Cette méthode extrait les informations sur tous les attributs d'une classe et les stocke dans un tableau, comme l'illustre le code suivant :

```
System.Reflection.MemberInfo typeInfo;
typeInfo = typeof(MyClass);
object [] attrs = typeInfo.GetCustomAttributes(false);
```

Vous pouvez ensuite parcourir le tableau afin de trouver les valeurs des attributs qui vous intéressent.



## Itération sur les attributs

Vous pouvez parcourir le tableau d'attributs par itération et examiner successivement la valeur de chaque attribut. Dans le code suivant, le seul attribut d'intérêt est **DeveloperInfoAttribute** ; tous les autres sont ignorés. Pour chaque attribut **DeveloperInfoAttribute** trouvé, les valeurs des propriétés **Developer** et **Date** sont affichées comme suit :

```
...
object [] attrs = typeInfo.GetCustomAttributes(false);
foreach(Attribute atr in attrs) {
 if (atr is DeveloperInfoAttribute) {
 DeveloperInfoAttribute dia = (DeveloperInfoAttribute)atr;
 Console.WriteLine("{0} {1}", dia.Developer, dia.Date);
 }
}
...
```

---

**Conseil** **GetCustomAttributes** est une méthode surchargée. Si vous voulez des valeurs pour ce seul type d'attribut, vous pouvez appeler cette méthode en passant le type de l'attribut personnalisé que vous recherchez par son intermédiaire, comme l'illustre le code suivant :

```
object [] attrs =
typeInfo.GetCustomAttributes(typeof(DeveloperInfoAttribute),
 false);
```

---

## Utilisation de la méthode IsDefined

En l'absence d'attributs correspondants pour une classe, **GetCustomAttributes** retourne une référence d'objet **null**. Toutefois, pour savoir si un attribut particulier a été défini pour une classe, vous pouvez utiliser la méthode **IsDefined** de **MemberInfo** comme suit :

```
Type devInfoAttrType = typeof(DeveloperInfoAttribute);
if (typeInfo.IsDefined(devInfoAttrType, false)) {
 object [] attrs =
 typeInfo.GetCustomAttributes(devInfoAttrType, false);
 ...
}
```

---

**Remarque** Vous pouvez utiliser le désassembleur ILDASM (Intermediate Language Disassembler) pour voir ces attributs à l'intérieur de l'assembly.

---

## Atelier 14.1 : Définition et utilisation d'attributs



---

### Objectifs

À la fin de cet atelier, vous serez à même d'effectuer les tâches suivantes :

- utiliser l'attribut **Conditional** prédéfini ;
- créer un attribut personnalisé ;
- ajouter une valeur d'attribut personnalisée à une classe ;
- utiliser la réflexion pour rechercher des valeurs d'attribut.

### Conditions préalables

Pour pouvoir aborder cet atelier, vous devez être familiarisé avec les éléments suivants :

- créer des classes en C# ;
- définir des constructeurs et des méthodes ;
- utiliser l'opérateur **typeof** ;
- utiliser des propriétés et des indexeurs en C#.

**Durée approximative de cet atelier : 45 minutes**

## Exercice 1

### Utilisation de l'attribut Conditional

Dans cet exercice, vous allez utiliser l'attribut **Conditional** prédéfini pour exécuter votre code de façon conditionnelle.

L'exécution conditionnelle est une technique pratique lorsque vous souhaitez incorporer du code de test ou de débogage dans un projet mais que vous ne voulez pas modifier le projet ou supprimer le code de débogage une fois le système terminé et opérationnel.

Au cours de cet exercice, vous allez ajouter une méthode appelée **DumpToScreen** à la classe **BankAccount** (créée dans des ateliers précédents). Cette méthode affichera les détails du compte. Vous utiliserez l'attribut **Conditional** pour exécuter cette méthode en fonction de la valeur d'un symbole appelé **DEBUG\_ACCOUNT**.

#### ► Pour appliquer l'attribut Conditional

1. Ouvrez le projet Bank.sln situé dans le dossier *dossier d'installation\Labs\Lab14\Starter\Bank*.
2. Dans la classe **BankAccount**, ajoutez une méthode void publique appelée **DumpToScreen** qui ne prend pas de paramètres.

La méthode doit afficher le contenu du compte : numéro de compte, titulaire du compte, type de compte et solde. Le code suivant montre une utilisation possible de la méthode :

```
public void DumpToScreen()
{
 Console.WriteLine("Débogage du compte {0}. Le titulaire
 ↪est {1}. Le type est {2}. Le solde est {3}".
 this.accNo, this.holder, this.accType, this.accBal);
}
```

3. Faites usage de la dépendance de la méthode vis-à-vis du symbole **DEBUG\_ACCOUNT**.

Ajoutez l'attribut **Conditional** suivant avant la méthode, comme suit :

```
[Conditional("DEBUG_ACCOUNT")]
```

4. Ajoutez une directive **using** pour l'espace de noms **System.Diagnostics**.
5. Compilez votre code et corrigez les erreurs, le cas échéant.

► **Pour tester l'attribut Conditional**

1. Ouvrez le projet TestHarness.sln situé dans le dossier *dossier d'installation\Labs\Lab14\Starter\TestHarness*.
2. Ajoutez une référence à la bibliothèque **Bank**.
  - a. Dans l'Explorateur de solutions, développez l'arborescence du projet **TestHarness**.
  - b. Cliquez avec le bouton droit sur **Références**, puis cliquez sur **Ajouter une référence**.
  - c. Cliquez sur **Parcourir**, puis accédez au *dossier d'installation\Labs\Lab14\Starter\Bank\Bin\Debug*.
  - d. Cliquez sur **Bank.dll**, sur **Ouvrir**, puis sur **OK**.
3. Passez en revue la méthode **Main** de la classe **CreateAccount**. Notez qu'elle crée un nouveau compte bancaire.
4. Ajoutez la ligne de code suivante dans **Main** pour appeler la méthode **DumpToScreen** de **myAccount** :  
`myAccount.DumpToScreen( );`
5. Enregistrez votre travail, compilez le projet et corrigez les erreurs, le cas échéant.
6. Exécutez le test de validation.  
 Notez que rien ne se produit. En effet, la méthode **DumpToScreen** n'a pas été appelée.
7. À partir de l'invite de commandes Visual Studio .NET, utilisez l'utilitaire ILDASM (ildasm) pour examiner *dossier d'installation\Labs\Lab14\Starter\Bank\Bin\Debug\Bank.dll*.  
 Notez que la méthode **DumpToScreen** est présente dans la classe **BankAccount**.
8. Double-cliquez sur la méthode **DumpToScreen** pour afficher le code MSIL (Microsoft Intermediate Language).  
 L'attribut **ConditionalAttribute** figure au début de la méthode. Le problème réside dans le test de validation. En raison du **ConditionalAttribute** sur **DumpToScreen**, le runtime ignore les appels passés à cette méthode si le symbole **DEBUG\_ACCOUNT** n'est pas défini lors de la compilation du programme appelant. L'appel est passé, mais comme **DEBUG\_ACCOUNT** n'est pas défini, le runtime met immédiatement fin à l'appel.
9. Fermez ILDASM.
10. Revenez au test de validation. En haut du fichier CreateAccount.cs, avant la première directive **using**, ajoutez le code suivant :  
`#define DEBUG_ACCOUNT`  
 Cette opération permet de définir le symbole **DEBUG\_ACCOUNT**.
11. Enregistrez et compilez le test de validation en corrigeant les erreurs, le cas échéant.
12. Exécutez le test de validation.  
 Notez que la méthode **DumpToScreen** affiche les informations de **myAccount**.

## Exercice 2

### Définition et utilisation d'un attribut personnalisé

Dans cet exercice, vous allez créer un attribut personnalisé appelé **DeveloperInfoAttribute**. Cet attribut permettra de stocker le nom du développeur et, éventuellement, la date de création d'une classe dans les métadonnées de cette classe. Cet attribut autorisera plusieurs utilisations, car plusieurs développeurs peuvent avoir participé au codage d'une classe.

Vous écrirez ensuite une méthode qui extrait et affiche toutes les valeurs **DeveloperInfoAttribute** d'une classe.

#### ► Pour définir une classe d'attribut personnalisé

1. Dans Visual Studio .NET, créez un nouveau projet Microsoft Visual C#, à l'aide des informations figurant dans le tableau ci-dessous.

| Élément        | Valeur                                            |
|----------------|---------------------------------------------------|
| Type de projet | Projets Visual C#                                 |
| Modèle         | Bibliothèque de classes                           |
| Nom            | CustomAttribute                                   |
| Emplacement    | <i>dossier d'installation</i> \Labs\Lab14\Starter |

2. Changez le nom et le nom de fichier de la classe **Class1** en **DeveloperInfoAttribute**.  
Veillez à changer également le nom du constructeur.
3. Spécifiez que la classe **DeveloperInfoAttribute** est dérivée de **System.Attribute**.  
Cet attribut sera applicable aux classes, enums et structs uniquement. Il sera également autorisé à apparaître plusieurs fois lors de son utilisation.
4. Ajoutez l'attribut **AttributeUsage** suivant avant la définition de la classe :  

```
[AttributeUsage(AttributeTargets.Class |
↳AttributeTargets.Enum | AttributeTargets.Struct,
↳AllowMultiple=true)]
```
5. Documentez votre attribut à l'aide d'un résumé explicite (entre les balises <summary>). Utilisez la description de l'exercice pour vous aider.
6. Pour l'attribut **DeveloperInfoAttribute**, le nom du développeur de la classe est un paramètre obligatoire et la date à laquelle la classe a été écrite est un paramètre facultatif de type string. Ajoutez des variables d'instance privées pour stocker ces informations, comme suit :  

```
private string developerName;
private string dateCreated;
```
7. Modifiez le constructeur de sorte qu'il n'accepte qu'un seul paramètre string également appelé **developerName**, et ajoutez une ligne de code dans le constructeur qui assigne ce paramètre à **this.developerName**.
8. Ajoutez une propriété **string** publique en lecture seule appelée **Developer** pouvant être utilisée pour obtenir (**get**) la valeur de **developerName**. N'écrivez pas d'accessor **set**.

9. Ajoutez une autre propriété **string** publique appelée **Date**. Cette propriété doit avoir un accesseur **get** qui lit **dateCreated** et un accesseur **set** qui écrit **dateCreated**.
10. Compilez la classe et corrigez les erreurs, le cas échéant.

Étant donné que la classe se trouve dans une bibliothèque de classes, le processus de compilation produit une DLL (CustomAttribute.dll) plutôt qu'un programme exécutable autonome. Le code complet de la classe **DeveloperInfoAttribute** est le suivant :

```
namespace CustomAttribute
{
 using System;
 /// <summary>
 /// Cette classe est un attribut personnalisé qui permet
 /// de stocker le nom du développeur d'une classe
 /// avec les métadonnées de cette classe.
 /// </summary>
 [AttributeUsage(AttributeTargets.Class |
 AttributeTargets.Enum | AttributeTargets.Struct,
 AllowMultiple=true)]
 public class DeveloperInfoAttribute: System.Attribute
 {
 private string developerName;
 private string dateCreated;

 // Constructeur. Le nom du développeur est le seul
 // paramètre obligatoire pour cet attribut.
 public DeveloperInfoAttribute(string developerName)
 {
 this.developerName = developerName;
 }
 public string Developer
 {
 get
 {
 return developerName;
 }
 }

 // Paramètre facultatif
 public string Date
 {
 get
 {
 return dateCreated;
 }
 set
 {
 dateCreated = value;
 }
 }
 }
}
```

► **Pour ajouter un attribut personnalisé à une classe**

1. Vous allez maintenant utiliser l'attribut **DeveloperInfo** pour enregistrer le nom du développeur de la classe de nombres **Rational**. (Cette classe a été créée dans un atelier précédent, mais elle vous est fournie ici pour plus de commodité.) Ouvrez le projet Rational.sln situé dans le dossier *dossier d'installation*\Labs\Lab14\Starter\Rational.
2. Exécutez les étapes suivantes pour ajouter une référence à la bibliothèque **CustomAttribute** que vous avez créée précédemment :
  - a. Dans l'Explorateur de solutions, développez l'arborescence du projet **Rational**.
  - b. Cliquez avec le bouton droit sur **Références**, puis cliquez sur **Ajouter une référence**.
  - c. Dans la boîte de dialogue **Ajouter une référence**, cliquez sur **Parcourir**.
  - d. Accédez au dossier *dossier d'installation*\Labs\Lab14\Starter\CustomAttribute\Bin\Debug, puis cliquez sur **CustomAttribute.dll**.
  - e. Cliquez sur **Ouvrir**, puis sur **OK**.
3. Ajoutez un attribut **CustomAttribute.DeveloperInfo** à la classe **Rational**, en spécifiant votre nom comme développeur et la date actuelle comme paramètre de date facultatif, comme suit :

```
[CustomAttribute.DeveloperInfo("Votre nom",
 ↪Date="Aujourd'hui")]
```
4. Ajoutez un deuxième développeur à la classe **Rational**.
5. Compilez le projet **Rational** et corrigez les erreurs, le cas échéant.
6. Ouvrez une fenêtre d'invite de commandes et accédez au dossier *dossier d'installation*\Labs\Lab14\Starter\Rational\Bin\Debug.  
Ce dossier doit contenir votre exécutable Rational.exe.
7. Exécutez ILDASM et ouvrez Rational.exe.
8. Développez l'espace de noms **Rational** dans l'arborescence.
9. Développez la classe **Rational**.
10. Près du haut de cette classe, remarquez votre attribut personnalisé et les valeurs que vous avez fournies.
11. Fermez ILDASM.

► **Pour utiliser la réflexion afin de rechercher des valeurs d'attribut**

Utiliser ILDASM est l'un des moyens d'examiner des valeurs d'attribut. Vous pouvez également utiliser la réflexion dans les programmes C#. Revenez dans Visual Studio .NET et modifiez la classe **TestRational** dans le projet **Rational**.

1. Dans la méthode **Main**, créez une variable appelée *attrInfo* de type **System.Reflection.MemberInfo**, comme l'illustre le code suivant :

```
public static void Main()
{
 System.Reflection.MemberInfo attrInfo;
 ...
}
```

2. Vous pouvez utiliser un objet **MemberInfo** pour stocker les informations sur les membres d'une classe. Assignez le type **Rational** à l'objet **MemberInfo** en utilisant l'opérateur **typeof**, comme suit :

```
attrInfo = typeof(Rational);
```

3. Les attributs d'une classe font partie des informations de la classe. Vous pouvez extraire les valeurs d'attribut en utilisant la méthode **GetCustomAttributes**. Créez un tableau d'objets appelé *attrs*, et utilisez la méthode **GetCustomAttributes** de *attrInfo* pour trouver tous les attributs personnalisés utilisés par la classe **Rational**, comme l'illustre le code suivant :

```
object[] attrs = attrInfo.GetCustomAttributes(false);
```

4. Vous devez maintenant extraire les informations d'attribut qui sont stockées dans le tableau *attrs* et les afficher. Créez une variable appelée *developerAttr* de type **CustomAttribute.DeveloperInfoAttribute**, et assignez-lui le premier élément du tableau *attrs* en effectuant le cast qui convient, comme illustré dans le code suivant :

```
CustomAttribute.DeveloperInfoAttribute developerAttr;
developerAttr =
 (CustomAttribute.DeveloperInfoAttribute)attrs[0];
```

---

**Remarque** Dans du code de production, vous utiliseriez la réflexion plutôt qu'un cast pour déterminer le type de l'attribut.

---

5. Utilisez l'accesseur **get** de l'attribut **DeveloperInfoAttribute** pour extraire les attributs **Developer** et **Date** et les afficher comme suit :

```
Console.WriteLine("Développeur : {0}\tDate : {1}",
 developerAttr.Developer, developerAttr.Date);
```

6. Répétez les étapes 4 et 5 pour l'élément 1 du tableau *attrs*.

Vous pouvez utiliser une boucle pour extraire les valeurs de plusieurs attributs.



7. Compilez le projet et corrigez les erreurs, le cas échéant.

Le code terminé de la méthode **Main** figure dans le code suivant :

```
namespace Rational
{
 using System;

 // Test de validation
 public class TestRational
 {
 public static void Main()
 {
 System.Reflection.MemberInfo attrInfo;
 attrInfo = typeof(Rational);
 object[] attrs = attrInfo.GetCustomAttributes(false);
 CustomAttribute.DeveloperInfoAttribute developerAttr;
 developerAttr =
 (CustomAttribute.DeveloperInfoAttribute)attrs[0];
 Console.WriteLine("Développeur : {0}\tDate : {1}",
 developerAttr.Developer, developerAttr.Date);
 developerAttr =
 (CustomAttribute.DeveloperInfoAttribute)attrs[1];
 Console.WriteLine("Développeur : {0}\tDate : {1}",
 developerAttr.Developer, developerAttr.Date);
 }
 }
}
```

Voici une méthode **Main** alternative qui utilise une boucle **foreach** :

```
public static void Main()
{
 System.Reflection.MemberInfo attrInfo;
 attrInfo = typeof(Rational);
 object[] attrs = attrInfo.GetCustomAttributes(false);

 foreach (CustomAttribute.DeveloperInfoAttribute
 devAttr in attrs)
 {
 Console.WriteLine("Développeur : {0}\tDate : {1}",
 devAttr.Developer, devAttr.Date);
 }
}
```

8. Lorsque vous exécutez ce programme, il affiche les noms et les dates que vous avez fournis en tant qu'informations **DeveloperInfoAttribute** à la classe **Rational**.

## Contrôle des acquis

- Vue d'ensemble des attributs
- Définition d'attributs personnalisés
- Extraction de valeurs d'attribut

- 
1. Pouvez-vous baliser des objets individuels à l'aide d'attributs ?
  2. Où les valeurs d'attribut sont-elles stockées ?
  3. Quel mécanisme est utilisé pour déterminer la valeur d'un attribut au moment de l'exécution ?

- 
4. Définissez une classe d'attribut appelée **CodeTestAttributes** qui ne s'applique qu'aux classes. Elle ne doit pas avoir de paramètre positionnel, mais deux paramètres nommés appelés **Reviewed** et **HasTestSuite**. Ces paramètres doivent être de type **bool** et être implémentés à l'aide de propriétés en lecture/écriture.
  
  5. Définissez une classe appelée **Widget** et utilisez **CodeTestAttributes** de la question précédente pour signaler que **Widget** a été passé en revue (**Reviewed**) mais qu'il n'a pas de **HasTestSuite**.
  
  6. Supposons que **Widget** de la question précédente disposait d'une méthode appelée **LogBug**. **CodeTestAttributes** pourrait-il être utilisé pour marquer uniquement cette méthode ?



---

## Annexe A : Ressources pour une étude approfondie

### Table des matières

Ressources pour C#

1



Les informations contenues dans ce document, notamment les adresses URL et les références à des sites Web Internet, pourront faire l'objet de modifications sans préavis. Sauf mention contraire, les sociétés, les produits, les noms de domaine, les adresses de messagerie, les logos, les personnes, les lieux et les événements utilisés dans les exemples sont fictifs et toute ressemblance avec des sociétés, produits, noms de domaine, adresses de messagerie, logos, personnes, lieux et événements existants ou ayant existé serait purement fortuite. L'utilisateur est tenu d'observer la réglementation relative aux droits d'auteur applicable dans son pays. Sans limitation des droits d'auteur, aucune partie de ce manuel ne peut être reproduite, stockée ou introduite dans un système d'extraction, ou transmise à quelque fin ou par quelque moyen que ce soit (électronique, mécanique, photocopie, enregistrement ou autre), sans la permission expresse et écrite de Microsoft Corporation.

Les produits mentionnés dans ce document peuvent faire l'objet de brevets, de dépôts de brevets en cours, de marques, de droits d'auteur ou d'autres droits de propriété intellectuelle et industrielle de Microsoft. Sauf stipulation expresse contraire d'un contrat de licence écrit de Microsoft, la fourniture de ce document n'a pas pour effet de vous concéder une licence sur ces brevets, marques, droits d'auteur ou autres droits de propriété intellectuelle.

© 2001–2002 Microsoft Corporation. Tous droits réservés.

Microsoft, MS-DOS, Windows, Windows NT, ActiveX, BizTalk, IntelliSense, JScript, MSDN, PowerPoint, SQL Server, Visual Basic, Visual C++, Visual C#, Visual J#, Visual Studio et Win32 sont soit des marques de Microsoft Corporation, soit des marques déposées de Microsoft Corporation, aux États-Unis d'Amérique et/ou dans d'autres pays.

Les autres noms de produit et de société mentionnés dans ce document sont des marques de leurs propriétaires respectifs.

## ◆ Ressources pour C#

- **Ouvrages sur la programmation en C#**
- **Ressources sur le développement en C#**
- **Ressources sur le développement avec le .NET Framework**

Cette annexe vous permet de rechercher les toutes dernières informations sur le langage de programmation C# et le Microsoft® .NET Framework. Elle mentionne des titres d'ouvrages et des adresses de sites Web qui intéresseront les développeurs, notamment :

- ouvrages sur la programmation en C# ;
- ressources sur le développement en C# ;
- ressources sur le développement avec le .NET Framework

## Ouvrages sur la programmation en C#

- **Microsoft Visual C# Étape par étape**
- ***A Programmer's Introduction to C#* (2ème édition, en anglais)**
- ***Inside C#* (en anglais)**
- ***C# Unleashed* (en anglais)**
- ***Programming C#* (en anglais)**
- ***C# Developer's Headstart* (en anglais)**
- ***C# and the .NET Platform* (en anglais)**

---

Les ouvrages suivants pourront vous aider à approfondir vos connaissances en programmation en C# :

- *Microsoft Visual C# Étape par étape*, de John Sharp et Jon Jagger, Microsoft Press®, 2002
- *A Programmer's Introduction to C# (2ème édition)*, d'Eric Gunnerson, Apress, 2001 (en anglais).
- *Inside C#*, de Tom Archer, Microsoft Press, 2001 (en anglais).
- *C# Unleashed*, de Joe Mayo, Sams Publishing, 2001 (en anglais).
- *Programming C#*, de Jesse Liberty, O'Reilly and Associates, 2001 (en anglais).
- *C# Developer's Headstart*, de Mark Michaelis et Philip Spokas, Osborne McGraw-Hill, 2001 (en anglais).
- *C# and the .NET Platform*, d'Andrew Troelsen, Apress, 2001 (en anglais).

Pour rechercher d'autres titres consacrés à la programmation en C#, consultez les sites Web suivants :

- <http://www.microsoft.com/france/mspress/produits/outils/vcsharp/>
- <http://www.microsoft.com/mspress/devtools/csharp>  
(en anglais) <http://www.dotnetbooks.com> (en anglais)



## Ressources sur le développement en C#

- <http://www.dotnetwire.com> (en anglais)
- <http://www.csharphelp.com> (en anglais)
- <http://www.csharp-station.com> (en anglais)
- <http://www.csharpindex.com> (en anglais)
- <http://www.codehound.com/csharp> (en anglais)
- <http://www.c-sharpcorner.com> (en anglais)

Les sites Web mentionnés ci-dessous proposent des informations précieuses sur le développement d'applications en C# :

- Pour accéder au principal site Web de Microsoft consacré à l'environnement .NET, rendez-vous à l'adresse : <http://www.dotnetwire.com> (en anglais)
- Pour consulter des articles, des informations et des commentaires sur C#, rendez-vous à l'adresse : <http://www.csharphelp.com> (en anglais).
- Pour prendre connaissance d'informations relatives à la programmation en C#, rendez-vous à l'adresse : <http://www.csharp-station.com> (en anglais)
- Pour avoir des informations de référence sur la programmation en C#, rendez-vous à l'adresse : <http://www.csharpindex.com> (en anglais)
- Pour accéder à un moteur de recherche C#, rendez-vous à l'adresse : <http://www.codehound.com/csharp> (en anglais)
- Pour accéder à un réseau de développeurs C# et .NET, rendez-vous à l'adresse : <http://www.c-sharpcorner.com> (en anglais)

## Ressources sur le développement avec le .NET Framework

### ■ Ressources

- <http://www.microsoft.com/france/net>
- <http://www.microsoft.com/france/msdn/technologies/technos/net>
- <http://www.gotnet.com> (en anglais)

### ■ Articles

---

Les sites Web Microsoft proposent des informations sur le développement de solutions pour le .NET Framework. Les sites suivants proposent des informations sur le langage de programmation C# :

- <http://www.microsoft.com/france/net/>
- <http://www.microsoft.com/france/msdn/technologies/technos/net/>
- <http://www.gotnet.com> (en anglais)

Les articles suivants (en anglais) proposent des informations plus approfondies sur le .NET Framework et sur les technologies qui s'y rapportent :

- « Microsoft .NET: Realizing the Next Generation Business Integration » à l'adresse : <http://www.microsoft.com/net/use/nextgenbiz.asp>
- « Microsoft .NET Framework FAQ » à l'adresse : <http://msdn.microsoft.com/library/techart/faq111700.htm>
- « C# Language Specification » à l'adresse : <http://msdn.microsoft.com/vstudio/nextgen/technology/csharpdownload.asp>
- « C# Introduction and Overview » à l'adresse : <http://msdn.microsoft.com/vstudio/nextgen/technology/csharpintro.asp>