Module 6: Lire et écrire des données locales

# **Contenu:**

|  |  |
| --- | --- |
|  | [Présentation du module](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C06.html#P1) |
| **Leçon 1:** | [Lecture et écriture de fichiers](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C06.html#P2) |
| **Leçon 2:** | [Sérialisation et désérialisation des données](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C06.html#P3) |
| **Lecon 3:** | [Exécution d'E / S à l'aide de flux](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C06.html#P4) |
| **Laboratoire:** | [Génération du rapport sur les notes](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C06.html#P5) |
|  | [Revue du module et points à retenir](https://www.skillpipe.com/api/2.1/content/urn:uuid:21ae3267-8517-4025-bf9a-2dd83635ed12@2020-12-11T20:16:12Z/OPS/html/20483C06.html#P6) |

# **Présentation du module**

La lecture et l'écriture de données sont des exigences essentielles pour de nombreuses applications, telles qu'un éditeur de fichier texte enregistrant un fichier dans le système de fichiers ou un service Windows écrivant des messages d'erreur dans un fichier journal personnalisé. Le système de fichiers local fournit l'environnement idéal pour une application pour lire et écrire de telles données, car l'accès est rapide et l'emplacement est facilement disponible. Microsoft .NET Framework fournit une variété de classes d'E / S qui simplifient le processus d'implémentation des fonctionnalités d'E / S dans vos applications.

Dans ce module, vous apprendrez à lire et à écrire des données à l'aide des opérations d'E / S du système de fichiers transactionnel, à sérialiser et désérialiser des données dans le système de fichiers et à lire et écrire des données dans le système de fichiers à l'aide de flux.

### **Objectifs**

Après avoir terminé ce module, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Lire et écrire des données vers et depuis le système de fichiers à l'aide des E / S de fichiers. |
| • | Convertissez les données dans un format qui peut être écrit ou lu à partir d'un fichier ou d'une autre source de données. |
| • | Utiliser des flux pour envoyer et recevoir des données vers ou depuis un fichier ou une source de données |

# Leçon 1: Lire et écrire des fichiers

Le .NET Framework fournit l'espace de noms System.IO, qui contient un certain nombre de classes qui simplifient les applications qui nécessitent des fonctionnalités d'E / S.

Dans cette leçon, vous apprendrez à utiliser les classes de cet espace de noms pour lire et écrire des données vers et à partir de fichiers, et pour manipuler des fichiers et des répertoires sur le système de fichiers.

## **Objectifs de la leçon**

Après avoir terminé cette leçon, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Lisez et écrivez des données à l'aide de la classe File. |
| • | Manipulez les fichiers à l'aide des classes FileInfo et File. |
| • | Manipulez les répertoires à l'aide des classes DirectoryInfo et Directory. |
| • | Manipulez les chemins de fichiers et de répertoires à l'aide de la classe Path. |

## **Lecture et écriture de données à l'aide de la classe File**

La classe File dans l'espace de noms System.IO expose plusieurs méthodes statiques que vous pouvez utiliser pour effectuer des opérations transactionnelles pour la lecture et l'écriture directes de fichiers. Ces méthodes sont transactionnelles car elles encapsulent plusieurs fonctions sous-jacentes dans un seul appel de méthode. En règle générale, pour lire les données d'un fichier, vous:

|  |  |
| --- | --- |
| 1. | Acquérir le descripteur de fichier. |
| 2. | Ouvrez un flux dans le fichier. |
| 3. | Mettez en mémoire tampon les données du fichier. |
| 4. | Relâchez le descripteur de fichier afin qu'il puisse être réutilisé. |

Les méthodes statiques exposées par la classe File sont pratiques car elles encapsulent des fonctions complexes de bas niveau. Cependant, leur commodité et le fait qu'ils protègent le développeur des fonctionnalités sous-jacentes signifient dans certains cas qu'ils n'offrent pas le contrôle ou la flexibilité dont les applications ont besoin. Par exemple, la méthode ReadAllText lira tout le contenu d'un fichier en mémoire. Pour les petits fichiers, ce sera bien, mais pour les fichiers volumineux, cela peut présenter des problèmes d'évolutivité et peut entraîner une interface utilisateur qui ne répond pas dans votre application.

**Lire des données à partir de fichiers**

La classe File fournit plusieurs méthodes que vous pouvez utiliser pour lire les données d'un fichier. Le format de vos données et la manière dont votre application entend les traiter influenceront la méthode à utiliser. La liste suivante décrit certaines de ces méthodes:

|  |  |
| --- | --- |
| • | La méthode ReadAllText vous permet de lire le contenu d'un fichier dans une seule variable de chaîne. L'exemple de code suivant montre comment lire le contenu du fichier settings.txt dans une chaîne nommée settings.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  paramètres de chaîne = File.ReadAllText (filePath); |
| • | La méthode ReadAllLines vous permet de lire le contenu d'un fichier et de stocker chaque ligne à un nouvel index dans un tableau de chaînes. L'exemple de code suivant montre comment lire le contenu du fichier settings.txt et stocker chaque ligne dans le tableau de chaînes nommé settingsLineByLine.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  string [] settingsLineByLine = File.ReadAllLines (filePath); |
| • | La méthode ReadAllBytes vous permet de lire le contenu d'un fichier sous forme de données binaires et de stocker les données dans un tableau d'octets. L'exemple de code suivant montre comment lire le contenu du fichier settings.txt dans un tableau d'octets nommé rawSettings.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  byte [] rawSettings = File.ReadAllBytes (filePath);  **Noter:**Pendant ces exemples et pour le reste de ce module, vous verrez la notation @ ””. Normalement, lorsque le caractère «\» apparaît dans une chaîne, il est traité comme un caractère d'échappement, transformant le caractère suivant en caractère Unicode ou ASCII spécial. Par exemple, la chaîne «\ n» sera transformée en caractère de nouvelle ligne. Pour écrire «\», vous devez en ajouter un autre pour l'échapper, comme ceci: «\\».  Les chaînes commençant par @, appelées chaînes textuelles, ne traitent pas «\» comme un caractère d'échappement, et tout ce qui y est écrit sera traité littéralement. Il est particulièrement utile de raccourcir les chemins, éliminant ainsi le besoin de plusieurs «\\».  Vous pouvez en savoir plus sur les chaînes ici: string (C # Reference),<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/string>.  Chacune de ces méthodes vous permet de lire le contenu d'un fichier en mémoire. Vous pouvez utiliser la méthode ReadAllText si vous souhaitez mettre en cache l'intégralité du fichier en mémoire en une seule opération. Sinon, si vous souhaitez traiter un fichier ligne par ligne, vous pouvez utiliser la méthode ReadAllLines pour lire chaque ligne dans un tableau.  **Écriture de données dans des fichiers**  La classe File fournit également des méthodes que vous pouvez utiliser pour écrire différents types de données dans un fichier. Pour chacun des différents types de données que vous pouvez écrire, la classe File propose deux méthodes: |
| • | Si le fichier spécifié n'existe pas, les méthodes Writexxx créent un nouveau fichier avec les nouvelles données. Si le fichier existe, les méthodes Writexxx écrasent le fichier existant par les nouvelles données. |
| • | Si le fichier spécifié n'existe pas, les méthodes Appendxxx créent également un nouveau fichier avec les nouvelles données. Cependant, si le fichier existe, les nouvelles données sont écrites à la fin du fichier existant. |

La liste suivante décrit certaines de ces méthodes:

|  |  |
| --- | --- |
| • | La méthode WriteAllText vous permet d'écrire le contenu d'une variable chaîne dans un fichier. Si le fichier existe, son contenu sera écrasé. L'exemple de code suivant montre comment écrire le contenu d'une chaîne nommée settings dans un nouveau fichier nommé settings.txt.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  string settings = "companyName = quatrième café;";  File.WriteAllText (filePath, paramètres); |
| • | La méthode WriteAllLines vous permet d'écrire le contenu d'un tableau de chaînes dans un fichier. Chaque entrée du tableau de chaînes représente une nouvelle ligne dans le fichier. L'exemple de code suivant montre comment écrire le contenu d'un tableau de chaînes nommé hosts dans un nouveau fichier nommé hosts.txt.  string filePath = @ "C: \ quatrièmeCoffee \ hosts.txt";  string [] hosts = {"86.120.1.203", "113.45.80.31", "168.195.23.29"};  File.WriteAllLines (filePath, hôtes); |
| • | La méthode WriteAllBytes vous permet d'écrire le contenu d'un tableau d'octets dans un fichier binaire. L'exemple de code suivant montre comment écrire le contenu d'un tableau d'octets nommé rawSettings dans un nouveau fichier nommé settings.txt.  string filePath = @ "C: \ quatrièmeCoffee \ setting.txt";  byte [] rawSettings = {99,111,109,112,97,110,121,78,97,109,101,61,102,111,  117, 114, 116, 104, 32, 99, 111, 102, 102, 101, 101};  File.WriteAllBytes (filePath, rawSettings); |
| • | La méthode AppendAllText vous permet d'écrire le contenu d'une variable chaîne à la fin d'un fichier existant. L'exemple de code suivant montre comment écrire le contenu d'une variable de chaîne nommée settings à la fin du fichier settings.txt existant.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  string settings = "companyContact = Dean Halstead";  File.AppendAllText (filePath, paramètres); |
| • | La méthode AppendAllLines vous permet d'écrire le contenu d'un tableau de chaînes à la fin d'un fichier existant. L'exemple de code suivant montre comment écrire le contenu d'un tableau de chaînes nommé newHosts dans le fichier hosts.txt existant. |

string filePath = @ "C: \ quatrièmeCoffee \ hosts.txt";

string [] newHosts = {"97.11.1.195", "203.194.40.177"};

File.WriteAllLines (filePath, newHosts);

Chacune de ces méthodes vous permet d'écrire des données dans un fichier. Si vous souhaitez ajouter des données à un fichier existant qui existe peut-être déjà, vous devez utiliser une méthode Appendxxx. Si vous souhaitez écraser un fichier existant, vous devez utiliser une méthode Writexxx. Ensuite, selon la façon dont vous souhaitez que les informations soient stockées (que ce soit sous forme de données binaires, d'un objet blob textuel dans une chaîne ou d'un tableau de chaînes représentant chaque ligne individuelle), utilisez la méthode xxxAllBytes, xxxAllText ou xxxAllLines.

## **Manipuler des fichiers**

Outre la lecture et l'écriture de fichiers, les applications nécessitent généralement la capacité d'interagir avec les fichiers stockés sur le système de fichiers. Par exemple, votre application peut avoir besoin de copier un fichier du répertoire système vers un emplacement temporaire avant d'effectuer un traitement supplémentaire, ou votre application peut avoir besoin de lire certaines métadonnées associées au fichier, telles que l'heure de création du fichier. Vous pouvez implémenter ce type de fonctionnalité à l'aide des classes File et FileInfo.

**Manipulation de fichiers à l'aide de la classe de fichiers**

La classe File fournit des méthodes statiques que vous pouvez utiliser pour effectuer une manipulation de fichier de base. La liste suivante décrit certaines de ces méthodes:

|  |  |
| --- | --- |
| • | La méthode Copy vous permet de copier un fichier existant dans un répertoire différent du système de fichiers. L'exemple de code suivant montre comment copier le fichier settings.txt du répertoire C: \ quatrièmeCoffee \ vers le répertoire C: \ temp \.  string sourceSettingsPath = @ "C: \ quatrièmeCoffee \ settings.txt";  string destinationSettingsPath = @ "C: \ temp \ settings.txt";  écrasement booléen = vrai;  File.Copy (sourceSettingsPath, destinationSettingsPath, overWrite);  **Noter:**Le paramètre overwrite passé à l'appel de la méthode Copy indique que le processus de copie doit écraser un fichier existant s'il existe dans le chemin de destination. Si vous passez false à l'appel de la méthode Copy et que le fichier existe déjà, le Common Language Runtime (CLR) lèvera une exception System.IO.IOException. |
| • | La méthode Delete vous permet de supprimer un fichier existant du système de fichiers. L'exemple de code suivant montre comment supprimer le fichier settings.txt existant.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  File.Delete (chemin\_fichier); |
| • | La méthode Exists vous permet de vérifier si un fichier existe sur le système de fichiers. L'exemple de code suivant montre comment vérifier si le fichier settings.txt existe.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  booléen persistedSettingsExist = File.Exists (filePath); |
| • | La méthode GetCreationTime vous permet de lire l'horodatage qui décrit le moment où un fichier a été créé, à partir des métadonnées associées au fichier. L'exemple de code suivant montre comment vous pouvez déterminer quand le fichier settings.txt a été créé.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  DateTime settingsCreatedOn = File.GetCreationTime (filePath);  Il existe de nombreuses autres opérations et métadonnées associées aux fichiers que vous pouvez utiliser dans vos applications. La classe FileInfo fournit un accès à ces derniers via un certain nombre de membres d'instance.  **Manipulation de fichiers à l'aide de la classe FileInfo**  La classe FileInfo fournit des membres d'instance que vous pouvez utiliser pour manipuler un fichier existant. Contrairement à la classe File qui fournit des méthodes statiques pour la manipulation directe, la classe FileInfo se comporte comme une représentation en mémoire du fichier physique, exposant les métadonnées associées au fichier via des propriétés et exposant des opérations via des méthodes.  L'exemple de code suivant montre comment créer une instance de la classe FileInfo qui représente le fichier settings.txt.  **Instanciation de la classe FileInfo**  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  Paramètres FileInfo = nouveau FileInfo (filePath);  Après avoir créé une instance de la classe FileInfo, vous pouvez utiliser les propriétés et les méthodes qu'elle expose pour interagir avec le fichier. La liste suivante décrit certaines de ces propriétés et méthodes: |
| • | La méthode CopyTo vous permet de copier un fichier existant dans un répertoire différent du système de fichiers. L'exemple de code suivant montre comment copier le fichier settings.txt du répertoire C: \ quatrièmeCoffee \ vers le répertoire C: \ temp \.  string sourceSettingsPath = @ "C: \ quatrièmeCoffee \ settings.txt";  string destinationSettingsPath = @ "C: \ temp \ settings.txt";  écrasement booléen = vrai;  Paramètres FileInfo = nouveau FileInfo (sourceSettingsPath);  settings.CopyTo (destinationSettingsPath, écraser);  **Noter:**Le paramètre overwrite passé à l'appel de méthode CopyTo indique que le processus de copie doit remplacer un fichier existant s'il existe dans le chemin de destination. Si vous passez false à l'appel de méthode CopyTo et que le fichier existe déjà, le CLR lèvera une exception System.IO.IOException. |
| • | La méthode Delete vous permet de supprimer un fichier. L'exemple de code suivant montre comment supprimer le fichier settings.txt.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  Paramètres FileInfo = nouveau FileInfo (filePath);  settings.Delete (); |
| • | La propriété DirectoryName vous permet d'obtenir le chemin du répertoire vers le fichier. L'exemple de code suivant montre comment obtenir le chemin d'accès au fichier settings.txt.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  Paramètres FileInfo = nouveau FileInfo (filePath);  string directoryPath = settings.DirectoryName; // renvoie C: \\ quatrièmeCoffee |
| • | La méthode Exists vous permet de déterminer si le fichier existe dans le système de fichiers. L'exemple de code suivant montre comment vérifier si le fichier settings.txt existe.  string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";  Paramètres FileInfo = nouveau FileInfo (filePath);  booléen persistedSettingsExist = settings.Exists; |
| • | La propriété Extension vous permet d'obtenir l'extension de fichier d'un fichier. L'exemple de code suivant montre comment obtenir l'extension d'un chemin renvoyé à partir d'un appel de méthode.  string filePath = FourthCoffeeDataService.GetDataPath ();  Paramètres FileInfo = nouveau FileInfo (filePath);  extension de chaîne = settings.Extension; |
| • | La propriété Length vous permet d'obtenir la longueur du fichier en octets. L'exemple de code suivant montre comment obtenir la longueur du fichier settings.txt. |

string filePath = @ "C: \ quatrièmeCoffee \ settings.txt";

Paramètres FileInfo = nouveau FileInfo (filePath);

longue longueur = paramètres.Longueur;

## **Manipulation des répertoires**

Il est courant que les applications interagissent et manipulent la structure des répertoires du système de fichiers, que ce soit pour vérifier qu'un répertoire existe avant d'écrire un fichier ou pour supprimer des répertoires lors de l'exécution d'un processus de nettoyage du système. La bibliothèque de classes .NET Framework fournit les classes Directory et DirectoryInfo pour ces opérations.

**Manipulation des répertoires à l'aide de la classe Directory**

Semblable à la classe File, la classe Directory fournit des méthodes statiques qui vous permettent d'interagir avec les répertoires, sans instancier un objet lié à un répertoire dans votre code. La liste suivante décrit certaines de ces méthodes statiques:

|  |  |
| --- | --- |
| • | La méthode CreateDirectory vous permet de créer un nouveau répertoire sur le système de fichiers. L'exemple suivant montre comment créer le répertoire C: \ quatrièmeCoffee \ tempData.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  Directory.CreateDirectory (directoryPath); |
| • | La méthode Delete vous permet de supprimer un répertoire à un chemin spécifique. L'exemple de code suivant montre comment supprimer le répertoire C: \ quatrièmeCoffee \ tempData et tout son contenu.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  booléen recursivelyDeleteSubContent = true;  Directory.Delete (directoryPath, recursivelyDeleteSubContent);  **Noter:**Le paramètre recursivelyDeleteSubContent passé dans l'appel de méthode Delete indique si le processus de suppression doit supprimer tout contenu pouvant exister dans l'annuaire. Si vous passez false dans l'appel de méthode Delete et que le répertoire n'est pas vide, le CLR lèvera une exception System.IO.IOException. |
| • | La méthode Exists vous permet de déterminer si un répertoire existe sur le système de fichiers. L'exemple de code suivant montre comment déterminer si le répertoire C: \ quatrièmeCoffee \ tempData existe.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  booléen tempDataDirectoryExists = Directory.Exists (directoryPath); |
| • | La méthode GetDirectories vous permet d'obtenir une liste de tous les sous-répertoires dans un répertoire spécifique sur le système de fichiers. L'exemple de code suivant montre comment obtenir une liste de tous les sous-répertoires dans le répertoire C: \ quatrièmeCoffee \ tempData.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  string [] subDirectories = Directory.GetDirectories (directoryPath); |
| • | La méthode GetFiles vous permet d'obtenir une liste de tous les fichiers dans un répertoire spécifique sur le système de fichiers. L'exemple suivant montre comment obtenir une liste de tous les fichiers du répertoire C: \ quatrièmeCoffee \ tempData.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  string [] files = Directory.GetFiles (directoryPath);  La classe DirectoryInfo fournit des membres d'instance qui vous permettent d'accéder aux métadonnées de l'annuaire et de manipuler la structure de l'annuaire.  **Manipulation des répertoires à l'aide de la classe DirectoryInfo**  La classe DirectoryInfo agit comme une représentation en mémoire d'un répertoire. Avant de pouvoir accéder aux propriétés et exécuter les méthodes exposées par la classe DirectoryInfo, vous devez créer une instance de la classe.  L'exemple de code suivant montre comment créer une instance de la classe DirectoryInfo qui représente le répertoire C: \ quatrièmeCoffee \ tempData.  **Instanciation de la classe DirectoryInfo**  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  DirectoryInfo directory = nouveau DirectoryInfo (directoryPath);  Lorsque vous avez créé une instance de la classe DirectoryInfo, vous pouvez ensuite utiliser ses propriétés et méthodes pour interagir avec l'annuaire. La liste suivante décrit certaines de ces propriétés et méthodes: |
| • | La méthode Create vous permet de créer un nouveau répertoire sur le système de fichiers. L'exemple suivant montre comment créer le répertoire C: \ quatrièmeCoffee \ tempData.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  DirectoryInfo directory = nouveau DirectoryInfo (directoryPath);  directory.Create (); |
| • | La méthode Delete vous permet de supprimer un répertoire à un chemin spécifique. L'exemple de code suivant montre comment supprimer le répertoire C: \ quatrièmeCoffee \ tempData et tout son contenu.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  booléen recursivelyDeleteSubContent = true;  DirectoryInfo directory = nouveau DirectoryInfo (directoryPath);  directory.Delete (recursivelyDeleteSubContent);  **Noter:**Le paramètre recursivelyDeleteSubContent passé à l'appel de méthode Delete indique si le processus de suppression doit supprimer tout contenu pouvant exister dans l'annuaire. Si vous passez false à l'appel de la méthode Delete et que le répertoire n'est pas vide, le CLR lèvera une exception System.IO.IOException. |
| • | La propriété Exists vous permet de déterminer si un répertoire existe sur le système de fichiers. L'exemple de code suivant montre comment déterminer si le répertoire C: \ quatrièmeCoffee \ tempData existe.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  DirectoryInfo directory = nouveau DirectoryInfo (directoryPath);  booléen tempDataDirectoryExists = directory.Exists; |
| • | La propriété FullName vous permet d'obtenir le chemin complet du répertoire. L'exemple suivant montre comment obtenir le chemin complet du répertoire tempData.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  DirectoryInfo directory = nouveau DirectoryInfo (directoryPath);  string fullPath = directory.FullName; |
| • | La méthode GetDirectories vous permet d'obtenir une liste de tous les sous-répertoires dans un répertoire spécifique sur le système de fichiers. Contrairement à la méthode statique File.GetDirectories, cette méthode d'instance renvoie un tableau de type DirectoryInfo, qui vous permet d'utiliser chacune des propriétés d'instance pour chaque sous-répertoire. L'exemple de code suivant montre comment obtenir tous les sous-répertoires du répertoire C: \ quatrièmeCoffee \ tempData.  string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";  DirectoryInfo directory = nouveau DirectoryInfo (directoryPath);  DirectoryInfo [] sous-répertoires = répertoire.GetDirectories (); |
| • | La méthode GetFiles vous permet d'obtenir une liste de tous les fichiers dans un répertoire spécifique sur le système de fichiers. Contrairement à la méthode statique File.GetFiles, cette méthode d'instance renvoie un tableau de type FileInfo, qui vous permet d'utiliser chacune des propriétés d'instance pour chaque fichier. L'exemple suivant montre comment obtenir tous les fichiers du répertoire C: \ quatrièmeCoffee \ tempData. |

string directoryPath = @ "C: \ quatrièmeCoffee \ tempData";

DirectoryInfo directory = nouveau DirectoryInfo (directoryPath);

FileInfo [] subFiles = répertoire.GetFiles ();

Selon que vous avez besoin d'une approche simple d'une ligne de code pour manipuler un répertoire, ou de quelque chose qui offre un peu plus de flexibilité, la classe statique Directory ou l'instance DirectoryInfo doit répondre à vos besoins.

## **Manipulation des chemins de fichiers et de répertoires**

Tous les fichiers et tous les répertoires ont un nom qui, lorsqu'il est combiné pour pointer vers un fichier dans un répertoire, constitue un chemin. Différents systèmes de fichiers peuvent avoir des conventions et des règles différentes pour ce qui constitue un chemin. Le .NET Framework fournit la classe Path, qui encapsule une variété de fonctions utilitaires de système de fichiers que vous pouvez utiliser pour analyser et construire des noms de fichiers, des noms de répertoires et des chemins valides dans le système de fichiers Windows. Ces fonctions peuvent être utiles si votre application a besoin d'écrire un fichier dans un emplacement temporaire, d'extraire un élément d'un chemin de système de fichiers ou même de générer un nom de fichier aléatoire.

Le code suivant montre comment créer un nouveau répertoire à la racine du lecteur C :.

**Créer un répertoire temporaire à la dure**

string tempDirectoryPath = @ "C: \ quatrièmeCafé \ tempData";

if (! Directory.Exists (tempDirectoryPath))

Directory.CreateDirectory (tempDirectoryPath);

Cependant, avec l'approche ci-dessus, vous faites de nombreuses hypothèses, notamment si votre application dispose des privilèges nécessaires pour effectuer des E / S à la racine du lecteur C et si le lecteur C existe réellement.

Une meilleure méthode consiste à utiliser la méthode statique GetTempPath fournie par la classe Path pour obtenir le chemin d'accès au répertoire temporaire Windows de l'utilisateur actuel.

**Obtention du chemin d'accès au répertoire temporaire de Windows**

string tempDirectoryPath = Path.GetTempPath ();

La classe Path comprend de nombreuses autres méthodes statiques qui fournissent un bon point de départ pour toute fonctionnalité de type d'E / S personnalisée dont votre application peut avoir besoin. Ces méthodes comprennent les suivantes:

|  |  |
| --- | --- |
| • | La méthode HasExtension vous permet de déterminer si le chemin que votre application traite a une extension. Cela vous permet de déterminer facilement si vous traitez un fichier ou un répertoire. L'exemple suivant montre comment vérifier si le chemin a une extension.  string settingsPath = ".. peut être n'importe quoi ici ..";  booléen hasExtension = Path.HasExtension (settingsPath); |
| • | La méthode GetExtension vous permet d'obtenir l'extension à partir d'un nom de fichier. Cette méthode est particulièrement utile lorsque vous souhaitez déterminer le type de fichier traité par votre application. L'exemple de code suivant montre comment vérifier si la variable settingsPath contient un chemin d'accès qui se termine par l'extension .txt.  string settingsPath = ".. peut être n'importe quoi ici ..";  string pathExt = Path.GetExtension (settingsPath);  if (pathExt == ".txt")  {  // Plus de traitement ici.  } |
| • | GetTempFileName vous permet de créer un nouveau fichier temporaire dans votre répertoire temporaire Windows local dans un seul dossier d'opérations transactionnelles. Cette méthode renvoie ensuite le chemin absolu vers ce fichier, prêt pour un traitement ultérieur. Le code suivant montre comment appeler la méthode GetTempFileName. |

string tempPath = Path.GetTempFileName ();

// Renvoie C: \ Users \ LeonidsP \ AppData \ Local \ Temp \ ABC.tmp

**Lecture supplémentaire:**Pour plus d'informations sur la classe Path, reportez-vous à la page Path Class à l'adresse

<https://aka.ms/moc-20483c-m6-pg1>.

## **Démonstration: manipulation de fichiers, de répertoires et de chemins**

Dans cette démonstration, vous utiliserez les classes File, Directory et Path pour créer un utilitaire qui combine plusieurs fichiers en un seul fichier.

### **Démonstration étapes**

Vous trouverez les étapes dans la section Démonstration: Manipuler des fichiers, des répertoires et des chemins sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_DEMO.md>.

# Leçon 2: Sérialisation et désérialisation des données

La sérialisation est le processus de conversion des données dans un format qui peut être conservé ou transporté. La désérialisation est le processus de conversion des données sérialisées en objets.

Dans cette leçon, vous apprendrez à sérialiser des objets en notation binaire, XML et JavaScript Object Notation (JSON), et à créer un sérialiseur personnalisé afin de pouvoir sérialiser des objets dans le format de votre choix.

## **Objectifs de la leçon**

Après avoir terminé cette leçon, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Décrivez le but de la sérialisation et les formats pris en charge par .NET Framework. |
| • | Créez un type personnalisé sérialisable. |
| • | Sérialisez un objet en tant que binaire. |
| • | Sérialisez un objet au format XML. |
| • | Sérialisez un objet en JSON. |
| • | Créez un sérialiseur personnalisé en implémentant l'interface IFormatter. |

## **Qu'est-ce que la sérialisation?**

Les applications traitent généralement les données. Les données sont lues dans la mémoire, peut-être à partir d'un fichier ou d'un appel de service Web, traitées, puis transmises à un autre composant du système pour un traitement ultérieur. Les composants d'un système peuvent fonctionner sur la même machine, mais généralement les composants fonctionnent sur différentes plates-formes, sur différents matériels et même dans des emplacements géographiques différents. Le format des données doit également être léger pour pouvoir être transporté via une variété de protocoles, tels que HTTP ou SOAP. La sérialisation est le processus de conversion de l'état d'un objet en une forme qui peut être persistante ou transportée.

**Formats sérialisables**

Les exigences de votre système et la manière dont vous prévoyez de transporter les données influenceront le format de sérialisation que vous choisissez. Le tableau suivant décrit certains des formats courants disponibles.

| **Format** | **La description** |
| --- | --- |
| Binaire | La sérialisation des données au format binaire lisible par machine vous permet de préserver la fidélité et l'état d'un objet entre les différentes instances de votre application. La sérialisation binaire est également rapide et légère, car le format binaire ne nécessite pas le traitement et le stockage de constructions de formatage inutiles.  La sérialisation binaire est couramment utilisée lors de la persistance et du transport d'objets entre des applications exécutées sur la même plate-forme.  Exemple binaire    100000010111010001101110010101110001111111010111000110011001011011100111100101101010010000100001101100010000000001111000001110001100000100000011000100101010001110110010110100001110010100010110101010011111101110101100001101011001101110110100100001111010100111100000101101111111010111000110011001011011100111100101101010010000100001101100010000000001111000001110001100000100000011000100101010001110110010110100001110010100010110101010011111101110101000110000010000001100010010101000111011001011010000 |
| XML | La sérialisation des données au format XML vous permet d'utiliser une norme ouverte qui peut être traitée par n'importe quelle application, quelle que soit la plate-forme. Contrairement au binaire, XML ne préserve pas la fidélité de type; il vous permet uniquement de sérialiser les membres publics que votre type expose.  Le format XML est plus détaillé que binaire, car les données sérialisées sont formatées avec des constructions XML. Cela rend le format XML moins efficace et plus gourmand en processeur pendant les processus de sérialisation, de désérialisation et de transport.  Cependant, la nature de XML en tant que texte brut et langage de forme libre permet aux messages d'être transmis à travers différentes applications et plates-formes. Tant que l'émetteur et le récepteur se sont mis d'accord sur un contrat connu, les deux peuvent envoyer et recevoir des messages et les convertir au modèle approprié dans leurs environnements respectifs.  La sérialisation XML était couramment utilisée pour sérialiser les données pouvant être transportées via le protocole SOAP vers et depuis les services Web. Cependant, en raison de la nature verbeuse et stricte de SOAP, ce protocole est tombé en disgrâce et ne se trouve généralement aujourd'hui que dans les environnements hérités.  Exemple XML SOAP    <SOAP-ENV: Enveloppe  xmlns: xsi = "http://www.w3.org/2001/XMLSchema-instance"  xmlns: xsd = "http://www.w3.org/2001/XMLSchema"  xmlns: SOAP-ENC = "http://schemas.xmlsoap.org/soap/encoding/"  xmlns: SOAP- ENV = http: //schemas.xmlsoap.org/soap/envelope/  xmlns: clr = "http://schemas.microsoft.com/soap/encoding/clr/1.0"  SOAP-ENV: encodingStyle = "http://schemas.xmlsoap.org/soap/encoding/">  <SOAP-ENV: Corps>  <a1: ServiceConfiguration id = "ref-1"  xmlns: a1 = "http://schemas.microsoft.com/clr/nsassem/  FourthCoffeeSerializer /% 2C% 20Version% 3D1.0.0.0% 2C% 20  Culture% 3Dneutral% 2C% 20PublicKeyToken% 3Dnull ">  <ConfigName id = "ref-3">  FourthCoffee\_Default  </ConfigName>  <DatabaseHostName id = "ref-4">  database209.fourthcoffee.com  </DatabaseHostName>  <ApplicationDataPath id = "ref-5">  C: \ quatrièmecoffee \ applicationdata \  </ApplicationDataPath>  </ a1: ServiceConfiguration>  </ SOAP-ENV: Corps>  </ SOAP-ENV: Enveloppe> |
| JSON | La sérialisation des données au format JSON vous permet d'utiliser un format d'échange de données léger basé sur un sous-ensemble du langage de programmation JavaScript. JSON est un format de texte simple qui est lisible par l'homme et également facile à analyser par machine, quelle que soit la plate-forme.  JSON partage les atouts XML sous forme de texte brut et de forme libre, ce qui en fait une plateforme multiplateforme. Cependant, contrairement à XML, il a une syntaxe beaucoup plus concise, ce qui le rend moins coûteux à transmettre et lisible par l'homme. Ces avantages ont fait de JSON le langage courant pour transmettre des données aujourd'hui, et la norme de facto dans l'industrie d'aujourd'hui.  JSON est couramment utilisé pour transporter des données entre tout, des appels JavaScript asynchrones et XML (AJAX) aux messages entre services Web, car contrairement à SOAP, vous n'êtes pas limité à une simple communication dans le même domaine.  Exemple JSON    {  "ConfigName": "FourthCoffee\_Default",  "DatabaseHostName": "database209.fourthcoffee.com",  "ApplicationDataPath": "C: \\ quatrième café \\ applicationdata \\"  } |

Sinon, si vous souhaitez sérialiser vos données dans un format que le .NET Framework ne prend pas en charge de manière native, vous pouvez implémenter votre propre classe de sérialiseur personnalisée.

## **Création d'un type sérialisable**

Le .NET Framework fournit de nombreuses classes sérialisables. Si vous souhaitez créer vos propres types qui sont sérialisables, vous devez vous assurer que la définition de type inclut la configuration et les fonctionnalités nécessaires pour que le serilalizer utilise. Le .NET Framework fournit les espaces de noms System et System.Runtime.Serialization, qui fournissent des classes pour activer la prise en charge de la sérialisation.

Pour créer un type sérialisable, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Définissez un constructeur par défaut.  ServiceConfiguration de classe publique  {  public ServiceConfiguration () {...}  } |
| 2. | Décorez la classe avec l'attribut Serializable fourni dans l'espace de noms System.  [Sérialisable]  ServiceConfiguration de classe publique  {  ...  } |
| 3. | Implémentez l'interface ISerializable fournie dans l'espace de noms System.Runtime.Serialization. La méthode GetObjectData vous permet d'extraire les données de votre objet pendant le processus de sérialisation.  [Sérialisable]  public class ServiceConfiguration: ISerializable  {  ...  **public void GetObjectData (informations sur SerializationInfo, contexte StreamingContext) {...}**  } |
| 4. | Définissez un constructeur de désérialisation, qui accepte les objets SerializationInfo et StreamingContext comme paramètres. Ce constructeur vous permet de réhydrater votre objet pendant le processus de désérialisation.  [Sérialisable]  public class ServiceConfiguration: ISerializable  {  ...  **public ServiceConfiguration (SerializationInfo info, StreamingContext ctxt) {...}**  } |
| 5. | Définissez les membres publics que vous souhaitez sérialiser. Vous pouvez demander au sérialiseur d'ignorer les champs privés en les décorant avec l'attribut NonSerialized. |

...

[Non sérialisé]

private Guid \_internalId;

chaîne publique ConfigName {get; ensemble; }

chaîne publique DatabaseHostName {get; ensemble; }

chaîne publique ApplicationDataPath {get; ensemble; }

...

L'exemple de code suivant montre la classe ServiceConfiguration complète, qui est sérialisable par l'une des implémentations IFormatter .NET Framework.

**Type sérialisable**

[Sérialisable]

public class ServiceConfiguration: ISerializable

{

[Non sérialisé]

private Guid \_internalId;

chaîne publique ConfigName {get; ensemble; }

chaîne publique DatabaseHostName {get; ensemble; }

chaîne publique ApplicationDataPath {get; ensemble; }

Service publicConfiguration ()

{

}

public ServiceConfiguration (SerializationInfo info, StreamingContext ctxt)

{

this.ConfigName

= info.GetValue ("ConfigName", typeof (chaîne)). ToString ();

this.DatabaseHostName

= info.GetValue ("DatabaseHostName", typeof (chaîne)). ToString ();

this.ApplicationDataPath

= info.GetValue ("ApplicationDataPath", typeof (chaîne)). ToString ();

}

public void GetObjectData (informations sur SerializationInfo, contexte StreamingContext)

{

info.AddValue ("ConfigName", this.ConfigName);

info.AddValue ("DatabaseHostName", this.DatabaseHostName);

info.AddValue ("ApplicationDataPath", this.ApplicationDataPath);

}

}

## **Sérialisation d'objets en tant que binaire**

Le .NET Framework fournit la classe BinaryFormatter dans l'espace de noms System.Runtime.Serialization.Formatters.Binary, que vous pouvez utiliser pour sérialiser et désérialiser des objets en tant que binaire.

**Noter:**Les classes BinaryFormatter et SoapFormatter implémentent l'interface IFormatter. Vous pouvez également implémenter l'interface IFormatter pour créer votre propre sérialiseur personnalisé.

**Sérialiser un objet à l'aide de la classe BinaryFormatter**

Pour sérialiser un objet à l'aide de la classe BinaryFormatter, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Obtenez une référence à l'objet que vous souhaitez sérialiser. |
| 2. | Créez une instance de BinaryFormatter que vous souhaitez utiliser pour sérialiser votre type. |
| 3. | Créez un flux que vous utiliserez comme tampon pour stocker les données sérialisées. |
| 4. | Appelez la méthode BinaryFormatter.Serialize, en transmettant le flux dans lequel les données sérialisées seront mises en mémoire tampon et l'objet que vous souhaitez sérialiser. |

L'exemple de code suivant montre comment utiliser la classe BinaryFormatter pour sérialiser un objet en tant que binaire.

**Exemple de sérialisation de BinaryFormatter**

// Créez l'objet que vous souhaitez sérialiser.

ServiceConfiguration config = ServiceConfiguration.Default;

// Créez le formateur que vous souhaitez utiliser pour sérialiser l'objet.

IFormatter formatter = new BinaryFormatter ();

// Crée le flux dans lequel les données sérialisées seront mises en mémoire tampon.

FileStream buffer = File.Create (@ "C: \ quatrièmecoffee \ config.txt");

// Appelle la méthode Serialize.

formatter.Serialize (tampon, configuration);

// Ferme le flux.

buffer.Close ();

Cet exemple sérialise l'objet ServiceConfiguration et écrit les données sérialisées dans un fichier. Il est important de noter que la sérialisation n'implique pas seulement l'écriture de données dans un fichier. La sérialisation est le processus de transformation d'un type dans un autre format, que vous pouvez ensuite écrire dans un fichier ou une base de données, ou envoyer via HTTP à un service Web.

**Désérialiser un objet à l'aide de la classe BinaryFormatter**

La désérialisation est le processus de transformation de votre objet sérialisé dans un format que votre application peut traiter. Pour désérialiser un objet à l'aide de la classe BinaryFormatter, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Créez une instance de BinaryFormatter que vous souhaitez utiliser pour désérialiser votre type. |
| 2. | Créez un flux pour lire les données sérialisées. |
| 3. | Appelez la méthode BinaryFormatter.Deserialize, en transmettant le flux qui contient les données sérialisées. |
| 4. | Convertissez le résultat de l'appel de méthode BinaryFormatter.Deserialize dans le type d'objet que vous attendez. |

L'exemple de code suivant montre comment utiliser la classe BinaryFormatter pour désérialiser des données binaires en un objet.

**Exemple de désérialisation de BinaryFormatter**

// Créez le formateur que vous souhaitez utiliser pour sérialiser l'objet.

IFormatter formatter = new BinaryFormatter ();

// Crée le flux dans lequel les données sérialisées seront également mises en mémoire tampon.

FileStream buffer = File.OpenRead (@ "C: \ quatrièmecoffee \ config.txt");

// Appel de la méthode Deserialize.

ServiceConfiguration config = formatter.Deserialize (buffer) as ServiceConfiguration;

// Ferme le flux.

buffer.Close ();

L'exemple ci-dessus lit les données binaires sérialisées à partir d'un fichier, puis désérialise le binaire dans un objet ServiceConfiguration. Le processus est le même pour la sérialisation et la désérialisation d'objets à l'aide de tous les formateurs qui implémentent l'interface IFormatter. Cela inclut la classe SoapFormatter et tous les formateurs personnalisés que vous pouvez implémenter.

## **Sérialisation d'objets au format XML**

Le .NET Framework fournit la classe SoapFormatter dans l'espace de noms System.Runtime.Serialization.Formatters.Soap, que vous pouvez utiliser pour sérialiser et désérialiser des objets au format XML.

**Sérialiser un objet à l'aide de la classe SoapFormatter**

Le processus de sérialisation des données au format XML est similaire au processus de sérialisation en binaire, à l'exception que vous utilisez la classe SoapFormatter.

L'exemple de code suivant montre comment utiliser la classe SoapFormatter pour sérialiser un objet au format XML.

**Exemple de sérialisation de SoapFormatter**

// Créez l'objet que vous souhaitez sérialiser.

ServiceConfiguration config = ServiceConfiguration.Default;

// Créez le formateur que vous souhaitez utiliser pour sérialiser l'objet.

IFormatter formatter = new SoapFormatter ();

// Crée le flux dans lequel les données sérialisées seront également mises en mémoire tampon.

FileStream buffer = File.Create (@ "C: \ quatrièmecoffee \ config.xml");

// Appelle la méthode Serialize.

formatter.Serialize (tampon, configuration);

// Ferme le flux.

buffer.Close ();

**Désérialiser un objet à l'aide de la classe SoapFormatter**

Le processus de désérialisation des données XML vers un objet est identique au processus de désérialisation des données binaires, à l'exception que vous utilisez la classe SoapFormatter.

L'exemple de code suivant montre comment utiliser la classe SoapFormatter pour désérialiser des données XML en un objet.

**Exemple de désérialisation de SoapFormatter**

// Créez le formateur que vous souhaitez utiliser pour sérialiser l'objet.

IFormatter formatter = new SoapFormatter ();

// Crée le flux dans lequel les données sérialisées seront également mises en mémoire tampon.

FileStream buffer = File.OpenRead (@ "C: \ quatrièmecoffee \ config.xml");

// Appel de la méthode Deserialize.

ServiceConfiguration config = formatter.Deserialize (buffer) as ServiceConfiguration;

// Ferme le flux.

buffer.Close ();

## **Sérialisation d'objets au format JSON**

Le .NET Framework prend également en charge la sérialisation des objets en tant que JSON à l'aide de la classe DataContractJsonSerializer dans l'espace de noms System.Runtime.Serialization.Json. Les étapes de sérialisation JSON sont différentes car la classe DataContractJsonSerializer est dérivée de la classe abstraite XmlObjectSerializer et ne constitue pas une implémentation de l'interface IFormatter.

**Sérialiser un objet à l'aide de la classe DataContractJsonSerializer**

Pour sérialiser un objet à l'aide de la classe DataContractJsonSerializer, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Obtenez une référence à l'objet que vous souhaitez sérialiser. |
| 2. | Créez une instance de la classe DataContractJsonSerializer que vous souhaitez utiliser pour sérialiser votre type. Le constructeur vous demande également de transmettre un objet Type, représentant le type d'objet que vous souhaitez sérialiser. |
| 3. | Créez un flux que vous utiliserez comme tampon pour stocker les données sérialisées. |
| 4. | Appelez la méthode DataContractJsonSerializer.WriteObject, en transmettant au flux que les données sérialisées seront également mises en mémoire tampon et l'objet que vous souhaitez sérialiser. |

L'exemple de code suivant montre comment utiliser la classe DataContractJsonSerializer pour sérialiser un objet en tant que JSON.

**Exemple de sérialisation de DataContractJsonSerializer**

// Créez l'objet que vous souhaitez sérialiser.

ServiceConfiguration config = ServiceConfiguration.Default;

// Créez un objet DataContractJsonSerializer que vous utiliserez pour sérialiser le

// objet vers JSON.

DataContractJsonSerializer jsonSerializer

= new DataContractJsonSerializer (config.GetType ());

// Crée le flux dans lequel les données sérialisées seront également mises en mémoire tampon.

FileStream buffer = File.Create (@ "C: \ quatrièmecoffee \ config.txt");

// Appel de la méthode WriteObject.

jsonSerializer.WriteObject (tampon, configuration);

// Ferme le flux.

buffer.Close ();

**Désérialiser un objet à l'aide de la classe DataContractJsonSerializer**

Pour désérialiser JSON en un objet à l'aide de la classe DataContractJsonSerializer, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Créez une instance de la classe DataContractJsonSerializer que vous souhaitez utiliser pour sérialiser votre type. Le constructeur vous oblige également à transmettre un objet Type, représentant le type d'objet que vous souhaitez désérialiser. |
| 2. | Créez un flux qui lira le JSON sérialisé en mémoire. |
| 3. | Appelez la méthode DataContractJsonSerializer.ReadObject, en passant le flux qui contient les données sérialisées. |
| 4. | Convertissez le résultat de l'appel de méthode DataContractJsonSerializer.ReadObject dans le type d'objet que vous attendez. |

L'exemple de code suivant montre comment utiliser la classe DataContractJsonSerializer pour désérialiser les données JSON en un objet.

**Exemple de désérialisation de DataContractJsonSerializer**

// Créez un objet DataContractJsonSerializer que vous utiliserez pour

// désérialise le JSON.

DataContractJsonSerializer jsonSerializer

= new DataContractJsonSerializer (typeof (ServiceConfiguration));

// Créer un flux qui lira les données sérialisées.

FileStream buffer = File.OpenRead (@ "C: \ quatrièmecoffee \ config.txt");

// Appel de la méthode ReadObject.

ServiceConfiguration config = jsonSerializer.ReadObject (buffer) as ServiceConfiguration;

// Ferme le flux.

buffer.Close ();

## **Sérialisation d'objets en tant que JSON à l'aide de JSON.Net**

Alors que le framework .NET fournit la classe DataContractJsonSerializer intégrée pour sérialiser des objets en JSON, la norme actuelle d'utilisation de JSON dans l'environnement .NET est la bibliothèque Json.NET de Newtonsoft. Json.NET fournit une bibliothèque étendue pour créer et interroger JSON. Le plus important pour nos besoins est leur classe JsonConvert qui peut sérialiser et désérialiser des classes vers et depuis JSON.

Pour utiliser la bibliothèque Json.Net, vous devrez l'ajouter à votre projet à l'aide de Nuget, qui est un gestionnaire de packages pour .Net, vous permettant de télécharger et de gérer facilement des bibliothèques tierces dans votre projet.

|  |  |
| --- | --- |
| 1. | Dans l'Explorateur de solutions, cliquez avec le bouton droit sur le projet. |
| 2. | Sélectionnez Gérer les packages Nuget. |
| 3. | Sélectionnez l'onglet Parcourir et recherchez JSON. |
| 4. | Sélectionnez le package Newtonsoft.Json, puis cliquez sur Installer. |

Contrairement aux implémentations .NET de l'interface IFormatter, JsonConvert ne nécessite pas que la classe de données soit décorée avec l'attribut Serializable. Toute classe peut être convertie en JSON sans traitement spécial. Cependant, Json.NET fournit un ensemble d'attributs pour spécifier exactement comment le modèle sera sérialisé, si nécessaire. Le plus utile d'entre eux est peut-être l'attribut JsonProperty, qui permet de déterminer le nom de la propriété dans la chaîne JSON sérialisée. Contrairement à Visual C #, la norme de dénomination de JSON pour les propriétés peut varier et suit généralement la convention lowerCamelCase, snake\_case ou même la convention kebab-case. Cette fonctionnalité peut être très utile, en particulier lors de la communication avec d'autres applications.

Un objet de données normal prêt à être utilisé avec JsonConvert peut ressembler à ceci:

L'exemple de code suivant montre la classe ServiceConfiguration complète, prête à être sérialisée avec Json.Net.

**Type sérialisable JSON**

ServiceConfiguration de classe publique

{

// JsonConvert ignore les membres privés par défaut

private Guid \_internalId;

// Mappe les propriétés avec les conventions de dénomination json

[JsonProperty ("configName")]

chaîne publique ConfigName {get; ensemble; }

[JsonProperty ("databaseHostName")]

chaîne publique DatabaseHostName {get; ensemble; }

[JsonProperty ("applicationDataPath")]

chaîne publique ApplicationDataPath {get; ensemble; }

}

**Sérialiser un objet à l'aide de la classe JsonConvert**

Pour sérialiser un objet à l'aide de la classe JsonConvert, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Obtenez une référence à l'objet que vous souhaitez sérialiser. |
| 2. | Appelez la méthode JsonConvert.Serialize. La méthode déduit automatiquement le type traité et le sérialise en conséquence. |

L'exemple de code suivant montre comment utiliser la classe JsonConvert pour sérialiser un objet en tant que JSON.

**Exemple de sérialisation de JsonConvert**

// Créez l'objet que vous souhaitez sérialiser.

ServiceConfiguration config = ServiceConfiguration.Default;

var jsonString = JsonConvert.Serialize (config);

**Désérialiser un objet à l'aide de la classe JsonConvert**

Pour désérialiser JSON en un objet à l'aide de la classe JsonConvert, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Obtenez la chaîne JSON dont vous avez besoin pour désérialiser. |
| 2. | Appelez la méthode JsonConvert.Desrialize <T>, en passant la chaîne à désérialiser, ainsi que le type cible en tant que paramètre de type générique. |

L'exemple de code suivant montre comment utiliser la classe JsonConvert pour désérialiser les données JSON en un objet.

**Exemple de désérialisation de JsonConvert**

// Récupère la chaîne JSON - Ici, nous supposons que c'est la même chose que dans l'exemple précédent.

// Désérialisation au type souhaité

var deserializedConfig = JsonConvert.DeserializeObject <ServiceConfiguration> (jsonString);

Vous avez peut-être remarqué que contrairement aux rubriques précédentes, nous sérialisons ici directement dans une chaîne, et non dans un flux, puis dans un fichier. Vous constaterez que la plupart des messages transmis sont suffisamment petits pour ne pas justifier l'utilisation d'un flux et les enregistrer dans une chaîne est parfaitement acceptable. Les ordinateurs modernes sont suffisamment puissants pour gérer des chaînes assez volumineuses.

Cependant, Json.Net permet la sérialisation vers et depuis les flux de manière très similaire aux types .Net internes, tout en permettant de supprimer l'interface ISerializable et en conservant l'utilisation laxiste de ses attributs par Json.Net.

**Sérialiser un objet à l'aide de la classe JsonSerializer**

Pour sérialiser un objet à l'aide de la classe JsonSerializer, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Obtenez une référence à l'objet que vous souhaitez sérialiser. |
| 2. | Créez une instance de la classe JsonSerializer que vous souhaitez utiliser pour sérialiser votre type. |
| 3. | Créez un écrivain de flux pour écrire l'objet. |
| 4. | Appelez la méthode JsonSerializer.Serialize, en transmettant à l'enregistreur de flux que les données sérialisées seront également mises en mémoire tampon et l'objet que vous souhaitez sérialiser. La méthode déduira automatiquement le type traité et le sérialisera en conséquence. |

L'exemple de code suivant montre comment utiliser la classe JsonSerializer pour sérialiser un objet en tant que JSON et l'enregistrer dans un fichier.

**Exemple de sérialisation de JsonSerializer**

// Créer le sérialiseur

var serializer = new JsonSerializer ();

// Ouvre le flux dans le fichier

var fileWriter = File.CreateText (@ "C: \ quatrièmecoffee \ config.json");

// Sérialiser et écrire l'objet dans le fichier

serializer.Serialize (fileWriter, config);

// Ferme le flux

fileWriter.Close ();

fileWriter.Dispose ();

**Désérialiser un objet à l'aide de la classe JsonSerializer**

Pour désérialiser JSON en un objet à l'aide de la classe JsonSerializer, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Créez une instance de la classe JsonSerializer que vous souhaitez utiliser pour sérialiser votre type. |
| 2. | Créez un flux, un lecteur de flux et JsonTextReader qui liront le JSON sérialisé dans la mémoire. |
| 3. | Appelez la méthode JsonSerializer.Deserialize <T>, en passant le JsonTextReader qui contient les données sérialisées, ainsi que le type cible en tant que paramètre de type générique. |
| 4. | Fermez tous les lecteurs et diffusez. |

L'exemple de code suivant montre comment utiliser la classe JsonSerializer pour désérialiser les données JSON en un objet.

**Exemple de désérialisation de JsonSerializer**

// Créer le sérialiseur

var serializer = new JsonSerializer ();

// Ouvre un flux dans le fichier

var fileReader = File.OpenRead (@ "C: \ quatrièmecoffee \ config.json");

// Créer un flux et des lecteurs de texte json

var textReader = new StreamReader (fileReader);

var jsonReader = new JsonTextReader (textReader);

// Désérialisation au type souhaité

var deserializedConfig = serializer.Deserialize <ServiceConfiguration> (jsonReader);

// Ferme tous les lecteurs et le flux

jsonReader.Close ();

textReader.Close ();

textReader.Dispose ();

fileReader.Close ();

fileReader.Dispose ();

**Lecture supplémentaire:**Pour plus d'informations sur la bibliothèque Json.Net, vous pouvez vous tourner vers les ressources suivantes:

Page d'accueil de Json.Net:<https://aka.ms/moc-20483c-m6-pg2>

Documentation Json.Net:<https://aka.ms/moc-20483c-m6-pg3>

Classe JsonConvert:<https://aka.ms/moc-20483c-m6-pg4>

Classe JsonPropertyAttribute:<https://aka.ms/moc-20483c-m6-pg5>

Classe JsonSerializer:<https://aka.ms/moc-20483c-m6-pg6>

## **Démonstration: sérialisation d'objets en tant que JSON à l'aide de JSON.Net**

Dans cette démonstration, vous verrez comment sérialiser et désérialiser des objets à l'aide de JSON.NET.

### **Démonstration étapes**

Vous trouverez les étapes dans la démonstration: Sérialisation d'objets en tant que JSON à l'aide de JSON.Net section sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_DEMO.md>.

## **Création d'un sérialiseur personnalisé**

Vous souhaiterez peut-être sérialiser les données dans un format autre que binaire, XML ou JSON. Le .NET Framework fournit l'interface IFormatter dans l'espace de noms System.Runtime.Serialization, afin que vous puissiez créer votre propre formateur. Votre formateur personnalisé suivra alors le même modèle que les classes BinaryFormatter et SoapFormatter.

Pour créer votre propre formateur, procédez comme suit:

|  |  |
| --- | --- |
| 1. | Créez une classe qui implémente l'interface IFormatter. |
| 2. | Créez des implémentations pour les propriétés SurrogateSelector, Binder et Context. |
| 3. | Créez des implémentations pour les méthodes Deserialize et Serialize. |

L'exemple de code suivant montre un formateur personnalisé qui peut sérialiser et désérialiser des objets au format .ini.

**IniFormatter personnalisé**

en utilisant le système;

using System.Collections.Generic;

en utilisant System.IO;

using System.Reflection;

using System.Runtime.Serialization;

espace de noms FourthCoffee.Serializer

{

classe IniFormatter: IFormatter

{

statique en lecture seule char [] \_delim = new char [] {'='};

public ISurrogateSelector SurrogateSelector {get; ensemble; }

public SerializationBinder Binder {get; ensemble; }

public StreamingContext Context {get; ensemble; }

public IniFormatter ()

{

this.Contexte

= nouveau StreamingContext (StreamingContextStates.All);

}

objet public Deserialize (Stream serializationStream)

{

Tampon StreamReader

= nouveau StreamReader (serializationStream);

// Récupère le type à partir des données sérialisées.

Tapez typeToDeserialize = this.GetType (buffer);

// Créer une instance d'objet par défaut en utilisant le nom de type.

Objet obj

= FormatterServices.GetUninitializedObject (typeToDeserialize);

// Récupère tous les membres du type.

MemberInfo [] membres

= FormatterServices.GetSerializableMembers (obj.GetType (), this.Context);

// Créer un dictionnaire pour stocker les noms de variables et toutes les données sérialisées.

Dictionnaire <chaîne, objet> serializedMemberData

= nouveau dictionnaire <chaîne, objet> ();

// Lire les données sérialisées et extraire les noms des variables

// et les valeurs sous forme de chaînes.

while (buffer.Peek ()> = 0)

{

string line = buffer.ReadLine ();

string [] sarr = line.Split (\_delim);

// clé = nom de la variable, valeur = valeur de la variable.

serializedMemberData.Add (

sarr [0] .Trim (), // Nom de la variable.

sarr [1] .Trim ()); // Valeur de la variable.

}

// Ferme le flux sous-jacent.

buffer.Close ();

// Crée une liste pour stocker les valeurs des membres sous leur type correct.

List <object> dataAsCorrectTypes = new List <object> ();

// Pour chacun des membres, récupère les valeurs sérialisées comme leur type correct.

for (int i = 0; i <members.Length; i ++)

{

FieldInfo field = members [i] as FieldInfo;

if (! serializedMemberData.ContainsKey (field.Name))

throw new SerializationException (field.Name);

// Changer le type de la valeur pour le type correct

// du membre.

object valueAsCorrectType = Convert.ChangeType (

serializedMemberData [field.Name],

field.FieldType);

dataAsCorrectTypes.Add (valueAsCorrectType);

}

// Remplit l'objet avec les valeurs désérialisées.

renvoie FormatterServices.PopulateObjectMembers (

obj,

membres,

dataAsCorrectTypes.ToArray ());

}

public void Serialize (Stream sérialisationStream, graphique d'objet)

{

// Récupère tous les champs que vous souhaitez sérialiser.

MemberInfo [] allMembers

= FormatterServices.GetSerializableMembers (graph.GetType (), this.Context);

// Récupère les données du champ.

object [] fieldData = FormatterServices.GetObjectData (graph, allMembers);

// Crée un tampon pour écrire également les données sérialisées.

StreamWriter sw = nouveau StreamWriter (serializationStream);

// Ecrit le nom de la classe sur la première ligne.

sw.WriteLine ("@ ClassName = {0}", graph.GetType (). FullName);

// Itère les données du champ.

pour (int i = 0; i <fieldData.Length; ++ i)

{

sw.WriteLine ("{0} = {1}",

allMembers [i] .Name, // Nom du membre.

fieldData [i] .ToString ()); // Valeur du membre.

}

sw.Close ();

}

private Type GetType (tampon StreamReader)

{

string firstLine = buffer.ReadLine ();

string [] sarr = firstLine.Split (\_delim);

chaîne nameOfClass = sarr [1];

return Type.GetType (nameOfClass);

}

}

}

# Leçon 3: Exécution d'E / S à l'aide de flux

Lorsque vous travaillez avec des données, qu'elles soient stockées dans un fichier sur le système de fichiers ou sur un serveur Web accessible via HTTP, les données deviennent parfois trop volumineuses pour être chargées en mémoire et transmises en une seule opération transactionnelle. Par exemple, imaginez essayer de charger un fichier vidéo de 200 gigaoctets du système de fichiers dans la mémoire en une seule opération. Non seulement l'opération prendrait beaucoup de temps, mais elle consommerait également une grande quantité de mémoire.

Dans cette leçon, vous apprendrez à utiliser des flux pour lire et écrire dans des fichiers sans avoir à mettre en cache l'intégralité du fichier en mémoire.

## **Objectifs de la leçon**

Après avoir terminé cette leçon, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Décrivez le but d'un flux. |
| • | Décrivez les différents types de flux fournis dans le .NET Framework. |
| • | Décrivez comment utiliser un flux. |

## **Que sont les flux?**

Le .NET Framework vous permet d'utiliser des flux. Un flux est une séquence d'octets, qui peut provenir d'un fichier sur le système de fichiers, d'une connexion réseau ou de la mémoire. Les flux vous permettent de lire ou d'écrire dans une source de données dans de petits paquets de données gérables. En règle générale, les flux fournissent les opérations suivantes:

|  |  |
| --- | --- |
| • | Lire des morceaux de données dans un type, tel qu'un tableau d'octets. |
| • | Écriture de morceaux de données d'un type dans un flux. |
| • | Interroger la position actuelle dans le flux et modifier une sélection spécifique d'octets à la position actuelle. |

**Streaming dans le .NET Framework**

Le .NET Framework fournit plusieurs classes de flux qui vous permettent de travailler avec une variété de données et de sources de données. Lors du choix des classes de flux à utiliser, vous devez tenir compte des éléments suivants:

|  |  |
| --- | --- |
| • | Quel type de données que vous lisez ou écrivez, par exemple, binaire ou alphanumérique. |
| • | Où les données sont stockées, par exemple, sur le système de fichiers local, en mémoire ou sur un serveur Web via un réseau. |

La bibliothèque de classes .NET Framework fournit plusieurs classes dans l'espace de noms System.IO que vous pouvez utiliser pour lire et écrire des fichiers à l'aide de flux. Au plus haut niveau d'abstraction, la classe Stream définit la fonctionnalité commune fournie par tous les flux. La classe fournit une vue générique d'une séquence d'octets, ainsi que les opérations et les propriétés fournies par tous les flux. En interne, un objet Stream gère un pointeur qui fait référence à l'emplacement actuel dans la source de données. Lorsque vous construisez pour la première fois un objet Stream sur une source de données, ce pointeur est positionné immédiatement avant le premier octet. Au fur et à mesure que vous lisez et écrivez des données, la classe Stream fait avancer ce pointeur vers la fin des données lues ou écrites.

Vous ne pouvez pas utiliser la classe Stream directement. Au lieu de cela, vous instanciez des spécialisations de cette classe qui sont optimisées pour effectuer des E / S basées sur des flux pour des types spécifiques de sources de données. Par exemple, la classe FileStream implémente un flux qui utilise un fichier disque comme source de données et la classe MemoryStream implémente un flux qui utilise un bloc de mémoire comme source de données.

## **Types de flux dans le .NET Framework**

Le .NET Framework fournit de nombreuses classes de flux que vous pouvez utiliser pour lire et écrire différents types de données à partir de différents types de sources de données. Le tableau suivant décrit certaines de ces classes de flux et quand vous voudrez peut-être les utiliser.

| **Classe de flux** | **La description** |
| --- | --- |
| **FileStream** | Vous permet d'établir un flux vers un fichier sur le système de fichiers. La classe FileStream gère les opérations telles que l'ouverture et la fermeture du fichier et fournit l'accès au fichier via une séquence brute d'octets. |
| **MemoryStream** | Vous permet d'établir un flux vers un emplacement de la mémoire. La classe MemoryStream gère des opérations telles que l'acquisition du stockage en mémoire et fournit un accès à l'emplacement mémoire via une séquence brute d'octets. |
| **NetworkStream** | Vous permet d'établir un flux vers un emplacement réseau en mémoire. La classe NetworkStream gère des opérations telles que l'ouverture et la fermeture d'une connexion à l'emplacement réseau et fournit l'accès à l'emplacement réseau via une séquence brute d'octets. |

Un flux établi à l'aide d'un objet FileStream, MemoryStream ou NetworkStream est simplement une séquence brute d'octets. Si les données source sont structurées, vous devez convertir la séquence d'octets dans les types appropriés. Cela peut être une tâche longue et source d'erreurs. Toutefois, le .NET Framework contient des classes que vous pouvez utiliser pour lire et écrire des données textuelles et des types primitifs dans un flux que vous avez ouvert à l'aide des classes FileStream, MemoryStream ou NetworkStream. Le tableau suivant décrit certaines des classes de lecture et d'écriture de flux.

| **Classe de flux** | **La description** |
| --- | --- |
| **StreamReader** | Vous permet de lire des données textuelles à partir d'un flux de source de données sous-jacent, tel qu'un objet FileStream, MemoryStream ou NetworkStream. |
| **StreamWriter** | Vous permet d'écrire des données textuelles dans un flux de source de données sous-jacent, tel qu'un objet FileStream, MemoryStream ou NetworkStream. |
| **BinaryReader** | Vous permet de lire des données binaires à partir d'un flux de source de données sous-jacent, tel qu'un objet FileStream, MemoryStream ou NetworkStream. |
| **BinaryWriter** | Vous permet d'écrire des données binaires dans un flux de source de données sous-jacent, tel qu'un objet FileStream, MemoryStream ou NetworkStream. |

## **Lecture et écriture de données binaires à l'aide de flux**

De nombreuses applications stockent des données sous forme binaire brute pour un certain nombre de raisons, telles que les suivantes:

|  |  |
| --- | --- |
| • | L'écriture de données binaires est rapide. |
| • | Les données binaires occupent moins d'espace sur le disque. |
| • | Les données binaires ne sont pas lisibles par l'homme. |

Vous pouvez lire et écrire des données au format binaire dans vos applications .NET Framework à l'aide des classes BinaryReader et BinaryWriter.

Pour lire ou écrire des données binaires, vous construisez un objet BinaryReader ou BinaryWriter en fournissant un flux qui est connecté à la source des données que vous souhaitez lire ou écrire, comme un objet FileStream ou MemoryStream.

L'exemple de code suivant montre comment initialiser les classes BinaryReader et BinaryWriter, en passant un objet FileStream.

**Initialisation d'un objet BinaryReader et BinaryWriter**

string filePath = @ "C: \ quatrièmecoffee \ applicationdata \ settings.txt";

FileStream file = nouveau FileStream (filePath);

...

Lecteur BinaryReader = nouveau BinaryReader (fichier);

...

Rédacteur BinaryWriter = nouveau BinaryWriter (fichier);

Après avoir créé un objet BinaryReader, vous pouvez utiliser ses membres pour lire les données binaires. La liste suivante décrit certains des membres clés:

|  |  |
| --- | --- |
| • | La propriété BaseStream vous permet d'accéder au flux sous-jacent utilisé par l'objet BinaryReader. |
| • | La méthode Close vous permet de fermer l'objet BinaryReader et le flux sous-jacent. |
| • | La méthode Read vous permet de lire le nombre d'octets restants dans le flux à partir d'un index spécifique. |
| • | La méthode ReadByte vous permet de lire l'octet suivant du flux et d'avancer le flux vers l'octet suivant. |
| • | La méthode ReadBytes vous permet de lire un nombre spécifié d'octets dans un tableau d'octets.  De même, l'objet BinaryWriter expose divers membres pour vous permettre d'écrire des données dans un flux sous-jacent. La liste suivante décrit certains des membres clés. |
| • | La propriété BaseStream vous permet d'accéder au flux sous-jacent utilisé par l'objet BinaryWiter. |
| • | La méthode Close vous permet de fermer l'objet BinaryWiter et le flux sous-jacent. |
| • | La méthode Flush vous permet de vider explicitement toutes les données de la mémoire tampon actuelle dans le flux sous-jacent. |
| • | La méthode Seek vous permet de définir votre position dans le flux actuel, écrivant ainsi dans un octet spécifique. |
| • | La méthode Write vous permet d'écrire vos données dans le flux et de faire avancer le flux. La méthode Write fournit plusieurs surcharges qui vous permettent d'écrire tous les types de données primitifs dans un flux. |

**Lecture de données binaires**

L'exemple de code suivant montre comment utiliser les classes BinaryReader et FileStream pour lire un fichier qui contient une collection d'octets. Cet exemple utilise la méthode Read pour avancer dans le flux d'octets du fichier.

**Exemple de BinaryReader**

// Chemin du fichier source.

chaîne sourceFilePath =

@ "C: \ quatrièmecoffee \ applicationdata \ settings.txt";

// Créer un objet FileStream pour pouvoir interagir avec le fichier

// système.

FileStream sourceFile = nouveau FileStream (

sourceFilePath, // Passez le chemin du fichier source.

FileMode.Open, // Ouvre un fichier existant.

FileAccess.Read); // Lit un fichier existant.

// Crée un objet BinaryWriter en passant l'objet FileStream.

Lecteur BinaryReader = nouveau BinaryReader (sourceFile);

// Stocke la position actuelle du flux.

position int = 0;

// Stocke la longueur du flux.

int length = (int) reader.BaseStream.Length;

// Crée un tableau pour stocker chaque octet du fichier.

byte [] dataCollection = nouvel octet [longueur];

int returnByte;

while ((returnByte = reader.Read ())! = -1)

{

// Définit la valeur à l'index suivant.

dataCollection [position] = (octet) retournéByte;

// Avance notre variable de position.

position + = taille de (octet);

}

// Ferme les flux pour libérer les descripteurs de fichiers.

reader.Close ();

sourceFile.Close ();

**Écriture de données binaires**

L'exemple de code suivant montre comment utiliser les classes BinaryWriter et FileStream pour écrire une collection d'entiers de quatre octets dans un fichier.

**Exemple de BinaryWriter**

string destinationFilePath = @ "C: \ quatrièmecoffee \ applicationdata \ settings.txt";

// Collection d'octets.

byte [] dataCollection = {1, 4, 6, 7, 12, 33, 26, 98, 82, 101};

// Créer un objet FileStream pour pouvoir interagir avec le fichier

// système.

FileStream destFile = nouveau FileStream (

destinationFilePath, // Passez le chemin de destination.

FileMode.Create, // Toujours créer un nouveau fichier.

FileAccess.Write); // N'effectue que l'écriture.

// Crée un objet BinaryWriter en passant l'objet FileStream.

Rédacteur BinaryWriter = nouveau BinaryWriter (destFile);

// Ecrit chaque octet dans le flux.

foreach (données d'octet dans dataCollection)

{

écrivain.Écrire (données);

}

// Ferme les deux flux pour vider les données dans le fichier.

écrivain.Close ();

destFile.Close ();

**Noter:**Il est préférable d'utiliser l'interface IDisposable avec une instruction using, plutôt que d'utiliser directement les méthodes Open et Close. Vous pouvez en apprendre plus à leur sujet dans le Module 11, Leçon 2, Sujet 2 - «Implémentation du modèle Dispose».

## **Lecture et écriture de données texte à l'aide de flux**

En plus de stocker des données sous forme de données binaires brutes, vous pouvez également stocker des données sous forme de texte brut. Vous souhaiterez peut-être effectuer cette opération dans votre application si les données persistantes doivent être lisibles par l'homme.

Le processus de lecture et d'écriture de texte brut dans un fichier est très similaire à la lecture et à l'écriture de données binaires, sauf que vous utilisez les classes StreamReader et StreamWriter.

Lorsque vous initialisez les classes StreamReader ou StreamWriter, vous devez fournir un objet de flux pour gérer l'interaction avec la source de données.

L'exemple de code suivant montre comment initialiser les classes StreamReader et StreamWriter, en passant un objet FileStream.

**Initialisation d'un objet StreamReader et StreamWriter**

string destinationFilePath = @ "C: \ quatrièmecoffee \ applicationdata \ settings.txt";

FileStream file = nouveau FileStream (destinationFilePath);

...

Lecteur StreamReader = nouveau StreamReader (fichier);

...

StreamWriter writer = nouveau StreamWriter (fichier);

Après avoir créé un objet StreamReader, vous pouvez utiliser ses membres pour lire le texte brut. La liste suivante décrit certains des membres clés:

|  |  |
| --- | --- |
| • | La méthode Close vous permet de fermer l'objet StreamReader et le flux sous-jacent. |
| • | La propriété EndOfStream vous permet de déterminer si vous avez atteint la fin du flux. |
| • | La méthode Peek vous permet d'obtenir le prochain caractère disponible dans le flux, mais ne le consomme pas. |
| • | La méthode Read vous permet d'obtenir et de consommer le prochain caractère disponible dans le flux. Cette méthode renvoie une variable int qui représente le binaire du caractère, que vous devrez peut-être convertir explicitement. |
| • | La méthode ReadBlock vous permet de lire un bloc entier de caractères à partir d'un index spécifique à partir du flux. |
| • | La méthode ReadLine vous permet de lire une ligne entière de caractères du flux. |
| • | La méthode ReadToEnd vous permet de lire tous les caractères à partir de la position actuelle dans le flux. |

De même, l'objet StreamWriter expose divers membres pour vous permettre d'écrire des données dans un flux sous-jacent. La liste suivante décrit certains des membres clés:

|  |  |
| --- | --- |
| • | La propriété AutoFlush vous permet d'indiquer à l'objet StreamWriter de vider les données dans le flux sous-jacent après chaque appel d'écriture. |
| • | La méthode Close vous permet de fermer l'objet StreamWriter et le flux sous-jacent. |
| • | La méthode Flush vous permet de vider explicitement toutes les données de la mémoire tampon actuelle dans le flux sous-jacent. |
| • | La propriété NewLine vous permet d'obtenir ou de définir les caractères utilisés pour les nouveaux sauts de ligne. |
| • | La méthode Write vous permet d'écrire vos données dans le flux et de faire avancer le flux. |
| • | La méthode WriteLine vous permet d'écrire vos données dans le flux suivi d'un nouveau saut de ligne, puis d'avancer le flux. |

Ces membres offrent de nombreuses options pour répondre à de nombreuses exigences différentes. Si vous ne souhaitez pas stocker l'intégralité du fichier en mémoire dans un seul bloc, vous pouvez utiliser une combinaison des méthodes Peek et Read pour lire chaque caractère, un à la fois. De même, si vous souhaitez écrire des lignes de texte dans un fichier une par une, vous pouvez utiliser la méthode WriteLine.

**Lecture de texte brut**

L'exemple de code suivant montre comment utiliser les classes StreamReader et FileStream pour lire un fichier contenant du texte brut. Cet exemple utilise la méthode Peak pour avancer dans le flux de caractères du fichier.

**Exemple StreamReader**

chaîne sourceFilePath =

@ "C: \ quatrièmecoffee \ applicationdata \ settings.txt";

// Créer un objet FileStream pour pouvoir interagir avec le fichier

// système.

FileStream sourceFile = nouveau FileStream (

sourceFilePath, // Passez le chemin du fichier source.

FileMode.Open, // Ouvre un fichier existant.

FileAccess.Read); // Lit un fichier existant.

Lecteur StreamReader = nouveau StreamReader (sourceFile);

StringBuilder fileContents = new StringBuilder ();

// Vérifie si la fin du fichier

// a été atteint.

while (reader.Peek ()! = -1)

{

// Lit le caractère suivant.

fileContents.Append ((char) reader.Read ());

}

// Stocke le contenu du fichier dans une nouvelle variable de chaîne.

string data = fileContents.ToString ();

// Fermez toujours les flux sous-jacents et libérez tous les descripteurs de fichiers.

reader.Close ();

sourceFile.Close ();

**Rédaction de texte brut**

L'exemple de code suivant montre comment utiliser les classes StreamWriter et FileStream pour écrire une chaîne dans un nouveau fichier sur le système de fichiers.

**Exemple StreamWriter**

chaîne destinationFilePath =

@ "C: \ quatrièmecoffee \ applicationdata \ settings.txt";

string data = "Bonjour, ceci sera écrit en texte brut";

// Créer un objet FileStream pour pouvoir interagir avec le fichier

// système.

FileStream destFile = nouveau FileStream (

destinationFilePath, // Passez le chemin de destination.

FileMode.Create, // Toujours créer un nouveau fichier.

FileAccess.Write); // N'effectue que l'écriture.

// Créer un nouvel objet StreamWriter.

StreamWriter writer = nouveau StreamWriter (destFile);

// Ecrit la chaîne dans le fichier.

writer.WriteLine (données);

// Fermez toujours les flux sous-jacents pour vider les données dans le fichier

// et relâchez tous les descripteurs de fichiers.

écrivain.Close ();

destFile.Close ();

## **Démonstration: Génération du laboratoire de rapport sur les notes**

Dans cette démonstration, vous découvrirez les tâches que vous effectuerez dans le laboratoire pour ce module.

### **Démonstration étapes**

Vous trouverez les étapes dans la section Démonstration: Génération du laboratoire de rapport de notes sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_DEMO.md>.

# **Atelier: Génération du rapport sur les notes**

### **Scénario**

Il vous a été demandé de mettre à niveau l'application Grades Prototype pour permettre aux utilisateurs d'enregistrer les notes d'un élève sous forme de fichier XML sur le disque local. L'utilisateur doit pouvoir cliquer sur un nouveau bouton dans la vue StudentProfile qui demande à l'utilisateur où il souhaite enregistrer le fichier, affiche un aperçu des données à l'utilisateur et demande à l'utilisateur de confirmer qu'il souhaite enregistrer le fichier. sur le disque. Si tel est le cas, l'application doit enregistrer les données de note au format XML à l'emplacement spécifié par l'utilisateur.

### **Objectifs**

Après avoir terminé cet atelier, vous serez en mesure de:

|  |  |
| --- | --- |
| • | Sérialisez les données dans un flux mémoire. |
| • | Désérialisez les données d'un flux mémoire. |
| • | Enregistrez les données sérialisées dans un fichier. |

##### **Configuration du laboratoire**

Durée estimée: 60 minutes

Vous trouverez les étapes de haut niveau sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_LAB_MANUAL.md>.

Vous trouverez les étapes détaillées sur la page suivante:<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_LAK.md>.

### **Exercice 1: Sérialisation des données du rapport sur les notes au format XML**

##### **Scénario**

Dans cet exercice, vous allez écrire du code qui s'exécute lorsque l'utilisateur clique sur le bouton Enregistrer le rapport dans la vue Profil de l'étudiant. Vous permettrez à un utilisateur de spécifier où enregistrer le rapport de notes et de sérialiser les données de notes afin qu'elles soient prêtes à être enregistrées dans un fichier.

Vous utiliserez l'objet SaveFileDialog pour demander à l'utilisateur le nom du fichier et l'emplacement où il souhaite enregistrer le fichier. Vous allez extraire les données de note de la source de données de l'application et les stocker dans une liste d'objets de note.

Vous écrirez ensuite la méthode FormatAsXMLStream. Cette méthode utilisera un objet XmlWriter pour créer un document XML et le remplir avec les informations de note de la liste des objets de note. Enfin, vous déboguerez l'application et afficherez les données contenues dans le flux mémoire.

**Résultat**: Une fois cet exercice terminé, les utilisateurs pourront spécifier l'emplacement du fichier de rapport de notes.

### **Exercice 2: Aperçu du rapport sur les notes**

##### **Scénario**

Dans cet exercice, vous allez écrire du code pour afficher un aperçu du rapport à l'utilisateur avant de l'enregistrer.

Tout d'abord, vous ajouterez du code à la méthode SaveReport\_Click pour afficher le document XML à l'utilisateur dans une boîte de message. Pour afficher le document, vous devez créer une représentation sous forme de chaîne du document XML stocké dans l'objet MemoryStream. Enfin, vous vérifierez que votre code fonctionne comme prévu en exécutant l'application et en prévisualisant le contenu d'un rapport.

**Résultat**: Une fois cet exercice terminé, les utilisateurs pourront prévisualiser un rapport avant de l'enregistrer.

### **Exercice 3: conservation des données de note sérialisées dans un fichier**

##### **Scénario**

Dans cet exercice, vous allez écrire les données de notation dans un fichier sur le disque local.

Vous commencerez par modifier la boîte de dialogue d'aperçu existante pour demander à l'utilisateur s'il souhaite enregistrer le fichier. S'ils souhaitent enregistrer le fichier, vous utiliserez un objet FileStream pour copier les données du MemoryStream dans un fichier physique. Ensuite, vous exécuterez l'application, générerez et enregistrerez un rapport, puis vérifierez que le rapport a été enregistré au bon emplacement dans le format correct.

**Résultat**: Une fois cet exercice terminé, les utilisateurs pourront enregistrer les rapports des élèves sur le disque dur local au format XML.

# **Revue du module et points à retenir**

Dans ce module, vous avez appris à travailler avec le système de fichiers à l'aide d'un certain nombre de classes dans l'espace de noms System.IO et à sérialiser les données d'application dans différents formats.

### **Question (s) de révision**