CAALS Thomas

CHRISTIAENS Mathilde

GROUSSEAU Thomas

JAROSSET Corentin

Cours C#

# Introduction

## Le langage C#

C# est un langage de programmation orienté objet[1](#_4.1._Langage_Orienté), commercialisé par Microsoft depuis 2002 et destiné à développer sur la plateforme Microsoft .NET.

Il est dérivé du C++ et très proche du Java dont il reprend la syntaxe générale ainsi que les concepts, y ajoutant des notions telles que la surcharge des opérateurs, les indexeurs et les délégués.

## .NET

Microsoft .NET (prononcé « dot net ») est le nom donné à un ensemble de produits et de technologies informatiques de l'entreprise Microsoft pour rendre des applications facilement portables sur Internet.

La plate-forme .NET se base sur plusieurs technologies :

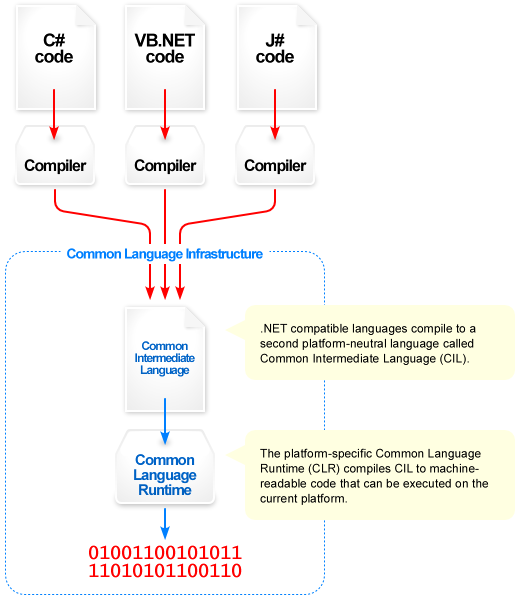
* Les systèmes d'exploitation propriétaires Microsoft Windows
* Le Framework .NET : un ensemble de bibliothèques de haut niveau
* Des protocoles de communication basés sur le Framework .NET
* Une bibliothèque logicielle compatible Framework .NET
* Un environnement d'exécution de code basé sur la CLI (Common Language Infrastructure)

## La CLI

Après compilation, comme en Java, le code C# est exécuté sur une machine virtuelle. La machine virtuelle du C# s’appelle la CLI, et ressemble beaucoup à la JVM (Java Virtual Machine) en Java.

Ces machines virtuelles agissent essentiellement comme une étape intermédiaire entre le code source du programmeur et le code machine du système, ce qui permet une plus grande facilité d'utilisation par divers types de processeurs. Dans l’ensemble, le fonctionnement de la JVM et de la CLI est pratiquement identique.

## Fonctionnement de la CLI

Le code en C#, J#, VB.NET ou autre langage .NET est compilé en un langage intermédiaire, le CIL (Common Intermediate Language), puis rentre dans la CLI.

Le CLR (Common Language Runtime) compile alors le CIL en code lisible par la machine, et l’application peut se lancer.

# Comparaison du C# et de java

Le langage C# a été au départ développé par Microsoft afin de concurrencer le langage Java, mais dans l’optique d’être intégré à l’initiative .NET. Il est donc compréhensible que les deux langages soient fortement semblables.

Les langages C# et Java sont très similaires au niveau du concept, et les différences que l’on peut noter entre les deux langages sont plutôt de l’ordre de l’implémentation de ces concepts.

## 2.1. C# vs Java : Similarités

### 2.1.1. Type-Safe :

Des erreurs de type peuvent se produire lorsque l’on essaie d’assigner une donnée d’un type à un objet d’un autre type, créant ainsi des effets secondaires non souhaités.

C# et Java font tous deux en sorte de s'assurer que les casts[2](#_4.2) non autorisés seront capturés au moment de la compilation et que les exceptions seront levées au moment de l'exécution si un type ne peut pas être converti en un autre type.

### 2.1.2. Garbage Collector :

Dans les langages de bas niveau, la gestion de la mémoire peut s'avérer fastidieuse car il faut se rappeler de supprimer correctement les objets qui ne seront plus utilisés pour libérer des ressources.

Ce n’est pas le cas en C# et en Java, où le garbage collector (ramasse-miette) permet d’éviter les fuites de mémoire en supprimant les objets qui ne seront plus utilisés par l’application. Bien que des fuites de mémoire puissent toujours se produire, les bases de la gestion de la mémoire sont déjà prises en charge.

### 2.1.3. Interfaces :

Une interface est une classe abstraite où toutes les méthodes sont abstraites. Une méthode abstraite est une méthode qui est déclarée mais ne contient pas les détails de son implémentation. Le code régissant les méthodes ou propriétés définies par l'interface doit être fourni par la classe qui l'implémente.

Une interface sera donc utilisée principalement pour regrouper certaines classes possédant des attributs en communs, et des fonctions qui seront appelées dans des cas similaires, mais qui font des actions différentes.

Par exemple : une interface mammifère qui regroupe les classes chats, chien, singe et souris. Ces classes ont des attributs en commun (dents, pattes, poils etc…) et des fonctions de même nom (manger, courir, crier etc…) mais dont l’action est différente.

### 2.1.4. Héritage unique :

C# et Java fonctionne sur un système d'héritage[4](#_4.4) unique - ce qui signifie qu'un seul chemin existe depuis une classe de base vers l'une de ses classes dérivées.

Toute classe C# ne peut donc hériter que d’une seule classe mais peut implémenter plusieurs interfaces, comme en Java.

Cela limite les effets secondaires inattendus pouvant survenir lorsqu'il existe plusieurs chemins entre plusieurs classes de base et classes dérivées. Le motif en losange[5](#_4.5) est un exemple classique de ce problème.

## 2.2. C# vs Java : Différences

### 2.2.1. Microsoft vs. Open-Source.

Même si des implémentations open-source existent, le C# est principalement utilisé pour développer sur la plateforme Microsoft – le CLR du Framework .NET étant l’implémentation la plus utilisée du CLI.

Au contraire, Java possède un écosystème open-source immense et a regagné un second souffle inattendu en partie grâce à Google qui a adopté la JVM pour Android.

### 2.2.2. Prise en charge des génériques.

Les génériques[6](#_4.6) améliorent grandement la vérification des types assistée par compilateur en supprimant les casts du code source.

En Java, les génériques sont implémentés en utilisant des effacements. Les paramètres de type générique sont « effacés » et les casts sont ajoutés lors de la compilation en bytecode.

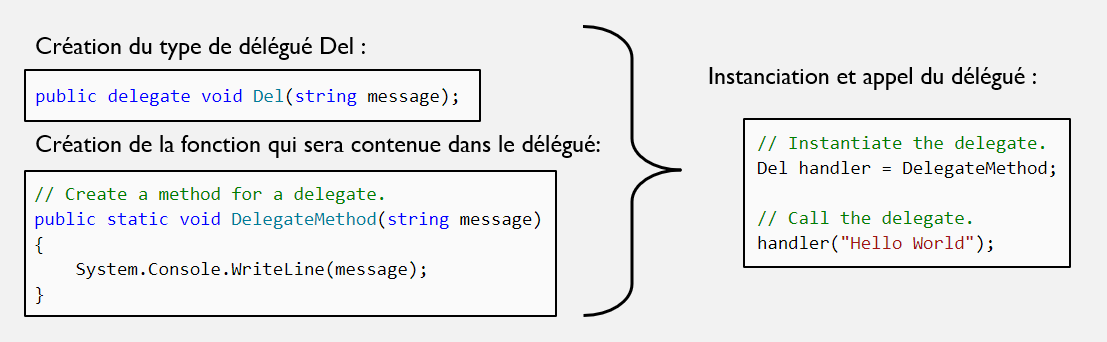
C # pousse les génériques encore plus loin en les intégrant dans la CLI et en permettant aux informations de type d'être disponibles au moment de l'exécution, générant un léger gain de performances.

### 2.2.3. Support pour les délégués.

C# a des délégués qui servent essentiellement de méthodes pouvant être appelées sans connaître l'objet cible. Il s’agit en fait de déclarer un type qui désigne une fonction, qui pourra ensuite être instancié avec des fonctions de mêmes paramètres et type de retour. Ces instances pourront ensuite être appelées de la même manière que les fonctions qu’elles contiennent.

Pour obtenir les mêmes fonctionnalités en Java, il faudrait utiliser une interface avec une seule méthode ou une autre solution de contournement pouvant nécessiter une quantité non négligeable de code supplémentaire, en fonction de l'application.

Utilisation des délégués :



### 2.2.4. Exceptions contrôlées :

Java distingue deux types d’exceptions[7](#_4.7) : contrôlées et non contrôlées. Les exceptions contrôlées sont vérifiées lors de la compilation, et empêche cette dernière si elles sont détectées. Les exceptions contrôlées ne sont pas détectées à la compilation et s’afficheront à l’exécution du programme (exemple : ArithmeticExeption).

C# a choisi une approche plus minimaliste en n’ayant qu’un seul type d’exception, les exceptions contrôlées. Si la capacité à intercepter des exceptions peut être utile, elle peut également avoir un impact négatif sur l’évolutivité et le contrôle de version.

### 2.2.5. Polymorphisme :

C# et Java adoptent des approches différentes du polymorphisme[8](#_4.8). Alors que Java active le polymorphisme par défaut, on doit utiliser en C# le mot clé *virtual* dans une classe de base et le mot clé *override* dans une classe dérivée.

### 2.2.6. Enumérations (Enums):

En C#, les énumérations[9](#_4.9) sont de simples listes de constantes nommées dont le type sous-jacent doit être intégral.

Java va plus loin avec l’enum en la traitant comme une instance nommée d'un type, ce qui facilite l'ajout d'un comportement personnalisé à des énumérations individuelles.

# Comparaison du C# et du C++

## 3.1. C# vs C++ : Similarités

### 3.1.1. Orienté objet :

Bien que la syntaxe soit légèrement différente, le concept des classes, l'héritage et le polymorphisme sont similaires en C++ et en C#.

### 3.1.2. Langages compilés :

Contrairement à Java qui est un langage interprété, C# et C++ sont des langages compilés. Cela signifie qu'avant le lancement d'une application, le code doit être converti en fichiers binaires.

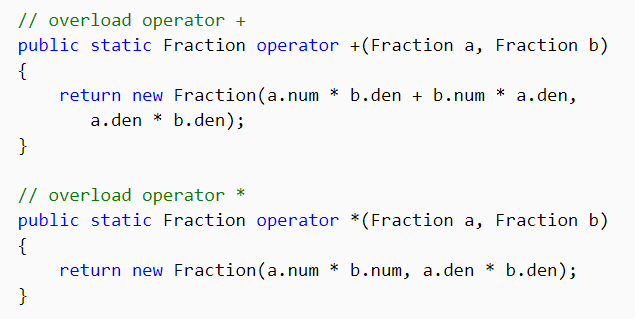
Un fichier exécutable est un exemple de fichier compilé pouvant être écrit en C++ ou en C #.

### 3.1.3. Surcharge d’opérateurs :

Le C# reprend le même système de surcharge d’opérateurs que celui qui existe en C++. Surcharger un opérateur, signifie soit modifier le code qui sera appelé lors de l’utilisation d’un opérateur (+, -, =, /, ++ etc…) sur un objet, soit ajouter des opérateurs à un objet.

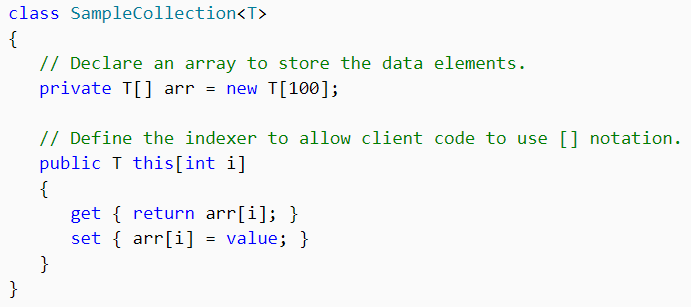
Par exemple, on pourrait vouloir créer une classe Date, qui aurait comme attribut un jour, un mois et une année, et vouloir additionner des instances de Date. Il faudrait donc pour cela surcharger l’opérateur + afin que l’instruction « date1 = date2 + date3 » additionne les dates en additionnant attribut par attribut et en comptant les retenues.

Sans cette surcharge d’opérateur, l’instruction « date1 = date2 + date3 » générerait simplement une erreur de compilation disant qu’il n’existe pas d’opérateur + pour des objets de type Date.

Exemple de surcharge des opérateurs + et \* pour une classe Fraction :

### 3.1.4. Les indexeurs :

Les indexeurs permettent aux instances d'une classe d'être indexés comme des tableaux. La valeur indexée peut être définie ou récupérée sans spécifier explicitement un membre de type ou d’instance. Les indexeurs ressemblent donc beaucoup à la surcharge d’opérateurs puisqu’ils surchargent l’opérateur [].

Exemple d’implémentation d’un indexeur :

## 3.2. C# vs C++ : Différences

### 3.2.1. Taille des fichiers binaires :

Comme mentionné précédemment, les deux langages sont des langages compilés qui transforment le code en fichiers binaires. C# a beaucoup de bibliothèques à inclure avant de compiler, bien plus que C++ qui par conséquent est beaucoup plus léger.

Autre conséquence : les fichiers binaires C# sont beaucoup plus volumineux après compilation que les fichiers binaires C++.

### 3.2.2. Performance :

C++ est largement utilisé lorsque les langages de niveau supérieur ne sont pas efficaces. Le code C++ est beaucoup plus rapide que le code C#, ce qui en fait une meilleure solution pour les applications où les performances sont importantes.

Le C# a en revanche d’autres avantages, dont principalement le fait d’être plus facile et pratique d’utilisation que le C++, étant un langage moins bas niveau.

Par exemple, un logiciel d'analyse de réseau peut avoir besoin de code C++, mais les performances ne constituent probablement pas un problème majeur pour une application de traitement de texte standard codée en C#.

### 3.2.3. Garbage collection :

Comme énoncé précédemment, C# (et plus précisément la CLR) comprend un système de garbage collector.

Ce garbage collector n’existe pas en C++, où il est nécessaire d’allouer et de désallouer la mémoire soi-même.

### 3.2.4. Plate-forme cible :

Les programmes C# sont généralement destinés au système d’exploitation Windows, bien que Microsoft s’emploie à prendre en charge le cross-platform pour les programmes C#, alors que le C++ permet de coder pour toute plate-forme, y compris Mac, Windows et Linux.

### 3.2.5. Types de projets :

Les programmeurs C++ se concentrent généralement sur des applications fonctionnant directement avec du matériel ou nécessitant de meilleures performances que celles offertes par d'autres langages.

Les programmes C++ incluent des applications côté serveur, des réseaux, des jeux et même des pilotes de périphérique pour votre PC.

C#, plus pratique principalement pour développer des applications graphiques, est généralement utilisé pour les applications Web, mobiles et de bureau.

### 3.2.6. Avertissements du compilateur :

Le compilateur du C++ permet de presque tout écrire, à condition que la syntaxe soit correcte. C’est un langage flexible, mais qui peut causer de sérieux dommages au système d’exploitation.

C# est beaucoup plus protégé et donne des erreurs de compilation et des avertissements et bloquera la plupart des erreurs sérieuses que C++ autorisera.

Pour reprendre les termes utilisés précédemment : les exceptions en C++ sont toutes non contrôlées, tandis que les exceptions du C# sont contrôlées

# Lexique

## 4.1. Langage Orienté Objet

Un langage orienté objet est un langage structuré dans un système de classe. Dans un langage orienté objet, une variable est obligatoirement un pointeur qui pointe vers une instance d’une classe.

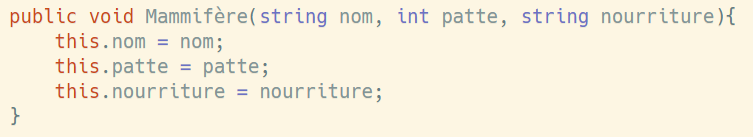
Classe VS Instance :

La classe est une définition d’un type. Ce type possède différentes propriétés :

- Des attributs : des variables internes au types. Par exemple une classe Mammifère aurait comme attribut entre autres le nombre de pattes (int), le nom de l’animal (string), le type de nourriture qu’il mange (string) etc… A l’intérieur de la classe, on accède aux attributs avec le mot-clé *this*.

- Des fonctions : des fonctions liées à la classes qui définissent ce que l’objet peut effectuer comme actions. Dans l’exemple du Mammifère, ses fonctions seraient manger, dormir, chasser etc…

Une classe possède également un constructeur, qui est une fonction de même nom que la classe, et qui prend généralement en argument les valeurs à entrer dans les attributs. Par exemple, la classe Mammifère, le constructeur serait la fonction suivante :



Une instance de classe est une variable du type de la classe. Par exemple, l’instance chat1 de la classe Mammifère est une variable de classe Mammifère déclarée par exemple comme ceci :

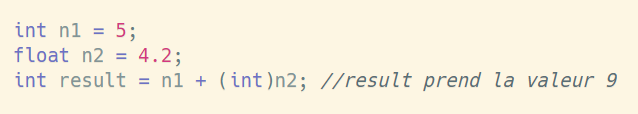


Dans ce cas, la classe Mammifère représente le groupe des mammifères, avec leur fonctionnement et les propriétés qu’ils peuvent avoir, et une instance de la classe Mammifère représente un mammifère en particulier, avec les propriétés qui lui sont propres.

## 4.2 Casts

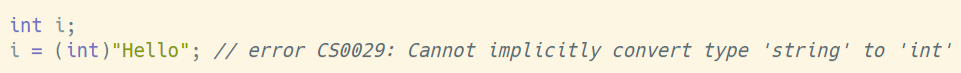
Un cast en programmation est l’action de forcer une variable à prendre un type différent.

Par exemple, si l’on veut ajouter une variable de type int et une autre de type float, il faut appliquer un cast sur l’un des deux, de la manière suivante :

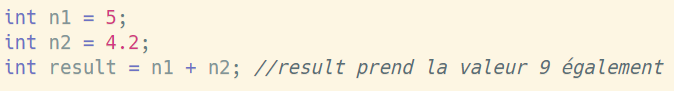


Dans ce cas précis, le cast de float vers int est possible, et est fait implicitement par le compilateur. Une partie de la donnée est cependant perdue : la partie décimale du nombre.

D’autres casts sont cependant impossibles et génèrent des erreurs :



Enfin, les casts implicites (comme le cast de float vers int) ne nécessitent pas de précéder la variable du type vers lequel le cast est fait. Ainsi, le code suivant fonctionnerait :



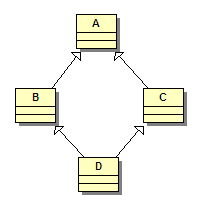
## 4.3 Héritage

L’héritage est un autre concept ajouté par la programmation objet. En orienté objet, une classe (que l’on appellera alors « classe fille » ou « classe dérivée ») peut hériter d’une classe (que l’on appelle donc « classe mère » ou « classe de base »). En héritant d’une classe mère, la classe fille hérite de ses propriétés.

Ainsi, la classe Chat, qui hérite de la classe Mammifère, aura également les attributs nom, pattes et nourriture, ainsi que les fonctions manger, dormir et chasser, sans avoir besoin de les réimplémenter. On pourra lui ajouter un attribut moustaches, par exemple ; et ainsi une instance de Chat possèdera un attribut qu’une instance de Mammifère n’aura pas, mais aura le reste de ses propriétés en commun.

## 4.4 Motif en losange

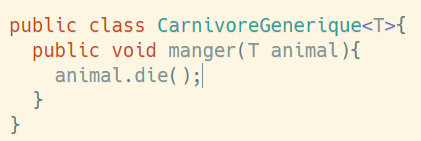
Le motif en losange est un cas particulier, impossible en C# et en Java mais pas en C++, dans lequel une classe hérite de deux classes qui elles-mêmes héritent de la même classe, selon le schéma suivant :

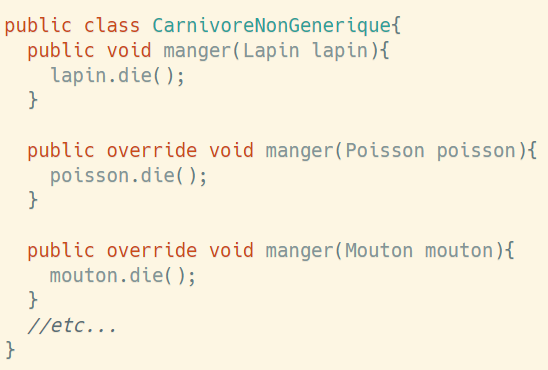


Cette implémentation peut générer de nombreux problèmes, principalement si une des classes B et C (ou les deux) a redéfini une des fonctions de la classe A (voir [Polymorphisme](#_4.7_Polymorphisme)).

## 4.5 Génériques

Un type générique permettent de concevoir des classes et des méthodes qui diffèrent la spécification d’un ou de plusieurs types jusqu’à ce que la classe ou la méthode soit déclarée et instanciée par le code client.

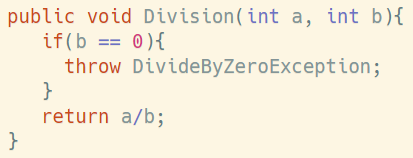
Par exemple, un type générique permet d’écrire une seule fonction capable de manipuler des arguments de types différents.

Ainsi la classe ci-dessus définit un type générique T qui remplace l’animal à manger dans la fonction manger(), ce qui permet de ne pas avoir l’implémentation suivante :

Dans cette implémentation sans générique, la fonction manger() a du être redéfinie une fois pour chaque type d’animal que le carnivore pourrait manger, au lieu d’avoir une fonction manger() qui peut manger n’importe quel animal (voir [Polymorphisme](#_4.8)).

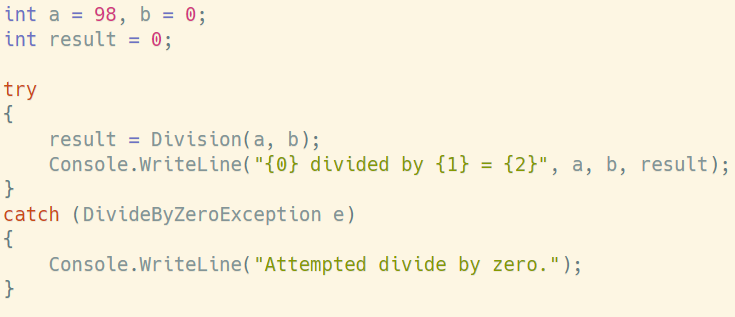
## 4.6 Exceptions

Une exception est une classe particulière qui dérive de la classe System.Exception. Des exceptions sont retournées lorsqu’une opération interdite est effectuée (l’exemple typique étant une division par 0). Une fonction qui renvoie une exception en cas d’opération interdite utilise le mot-clé *throw*. Ci-dessous l’exemple d’une fonction de division qui renvoie *DivideByZeroException* en cas de diviseur nul :



Une exception peut être « attrapée » à la compilation grâce aux instructions *try* et *catch*. Dans un bloc *try*, on essaie d’exécuter des instructions. Si une exception est renvoyée à l’intérieur du bloc *try*, les instructions qui suivent ne sont pas exécutées, et le code passe dans le bloc catch. Dans ce bloc, on pourra par exemple écrire en console un message à l’utilisateur expliquant pourquoi le code s’est arrêté.

Par exemple pour attraper l’exception renvoyée par la fonction Division ci-dessus :



Ici, on appelle la fonction avec un diviseur égal à 0. La fonction renvoie donc l’exception *DivideByZeroException*, qui est ensuite attrapée. Le message qui sera écrit en console ne sera donc pas « 98 divided by 0 = … » mais « Attempted divide by zero. »

## 4.7 Polymorphisme

Le polymorphisme possède deux aspects :

* D’un côté, il s’agit de redéfinir des fonctions définies auparavant. Le polymorphisme peut s’effectuer sur des fonctions définies précédemment dans la classe, principalement pour avoir plusieurs fonctions de même nom mais avec un code et des arguments différents.

Ce type de polymorphisme est plus généralement utilisé afin de redéfinir dans une classe fille des fonctions de la classe mère (ou d’une des classes dont elle dérive). Cette fois-ci, la fonction a le même nom et les mêmes arguments, mais un code différent, et contient dans sa définition le mot-clé *override*.

Certaines fonctions appelées virtuelles ne contiennent pas de code et ont pour objectif d’être redéfinies dans les classes dérivées. Une classe qui contient au moins une fonction virtuelle est appelée classe virtuelle (la classe et la/les fonction(s) virtuelle(s) doivent contenir le mot-clé *virtual* dans leur définition).

* De l’autre côté, il s’agit de considérer une instance d’une classe comme étant une instance d’une des classes dont elle dérive. Par exemple, si la classe Chat a redéfini la fonction manger() de Mammifère et qu’on appelle cette fonction de l’instance de Chat en faisant un cast vers Mammifère, le code qui sera exécuté sera celui de la fonction manger de Mammifère.

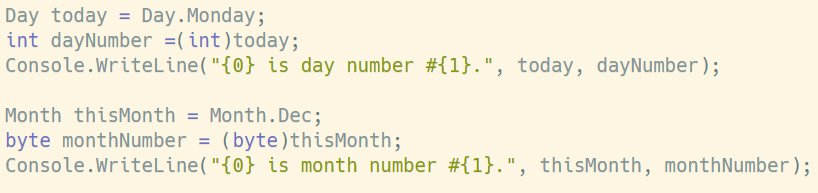
De même, une fonction qui prend une instance de Mammifère en paramètre pourra également prendre une instance de Chat en paramètre sans générer d’erreurs (et ici le cast est automatique et implicite).

## 4.8 Enumérations

Une énumération est un type de classe particulier qui n’a pour intérêt que de stocker un nombre défini de valeurs, elle-même définies.



Dans l’exemple ci-dessus, on a une première énumération Day qui contient les jours de la semaine et une énumération Month qui contient les mois de l’année. Le type byte sur cette énumération, signifie que les mois peuvent également être liés à leur position dans l’énumération, retournée sous la forme d’un byte.



Ainsi, le code ci-dessus affichera « Monday is day number #1 » et « Dec is month number #11 ».