Chapitre 3
Programmation orientée objet

1. Principes de la programmation orientée objet

La programmation orientée objet (POO) est un paradigme très répandu en développement logiciel. Il vient compléter un panorama déjà riche du paradigme procédural ainsi que du paradigme fonctionnel.

La POO est une forme de conception de code visant à représenter les données et les actions comme faisant partie de classes, elles-mêmes devenant des objets lors de leur création en mémoire. Cette notion a été rapidement présentée dans le chapitre précédent, il est maintenant temps de comprendre son fonctionnement plus en détail.

1.1 Qu'est-ce qu'une classe ?

Une classe est un élément du système que forme votre application. Une classe contient deux types d'élément de code : des données ainsi que des méthodes, représentant des actions. Il faut voir la classe comme étant une boîte dans laquelle il est possible de ranger ces deux types d'éléments. Pour faire un parallèle avec la vie réelle, nous pouvons facilement comprendre que la définition d'une classe s'applique à un objet comme un ordinateur, par exemple. Ce dernier dispose de méthodes (allumer, éteindre...) ainsi que des propriétés (nombre d'écrans, quantité de RAM...).

Conceptuellement, une classe n'est qu'une définition. Une fois que vous avez statué sur ce qu'elle doit contenir ainsi que ses méthodes, il convient de la créer. Cette action s'appelle l'instanciation. À la suite de cette opération, nous obtenons une instance en mémoire d'un objet.

Pour tenter une comparaison, prenons l'exemple d'une usine de fabrication d'objets en bois. Afin de pouvoir créer un objet, il faut un plan (la classe). Grâce à ce dernier, la machine peut découper et assembler les divers éléments (les données et méthodes) afin de créer une nouvelle instance (instanciation).

En C#, la déclaration d'une classe se fait grâce au mot-clé `class`. Il y a quelques spécificités possibles, notamment la portée, que nous étudierons juste après, dans la section Que peut-on déclarer dans une classe ? - Les méthodes, ainsi que les concepts de `static`, `sealed` et celui de `partial`. La syntaxe complète de la déclaration d'une classe est la suivante :

```
PORTÉE [static] [sealed] [partial] class NOM_CLASSE
```

Le nom de la classe est libre mais répond à deux règles :

– Il ne peut contenir que des caractères alphanumériques et le signe underscore (« _ »).

– Il ne peut pas commencer par un chiffre.

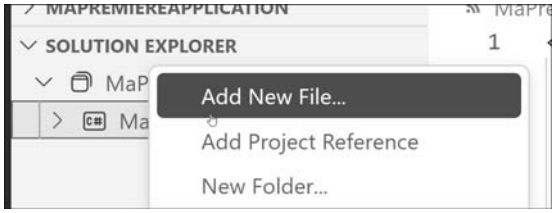
En plus de ces règles, les développeurs C# respectent souvent une convention syntaxique : l'utilisation du PascalCase. Cela indique que le nom commence par une majuscule et chaque mot est symbolisé par une majuscule également, par exemple, `OrdinateurPortable`. Le langage et le compilateur n'interdisent pas d'écrire `ordinateurPortable`, `Ordinateurportable` ou encore `ordinateurportable`, mais ces différentes déclarations ne respectent pas la convention largement admise et appliquée. Finalement, bien que ce soit possible, il est recommandé d'éviter tout caractère accentué dans le nom d'une classe. Par exemple, il est préférable d'appeler sa classe `Pieton` plutôt que `Piéton`, cela afin que le code C# produit soit le plus proche possible de ce que nous aurions en anglais.

Dans le programme de base créé en C# dans le précédent chapitre, une classe `Program` est créée par défaut. Nous pouvons constater qu'il n'y a ni notion de portée ni notion de `partial` ou `static`. Une fois qu'une classe est déclarée, elle définit un bloc, dans lequel nous pouvons implémenter les données et les méthodes dont notre programme a besoin pour fonctionner.

1.1.1 Les classes dans Visual Studio Code

Pour créer une classe dans Visual Studio Code, il est nécessaire de suivre les étapes suivantes :

- Dépliez la vue **Solution Explorer** du projet.
- Placez-vous sur le dossier où vous souhaitez créer la nouvelle classe (ou directement sur le nom du projet si vous souhaitez la créer à la racine).
- Faites un clic droit pour sélectionner l'élément de menu **Add New File**.
- Renseignez le nom de la classe dans la petite pop-up qui s'est ouverte en haut au centre de l'écran (toujours sans espaces ni caractères spéciaux).



Ajout d'une nouvelle classe avec Visual Studio Code

À la suite de ces manipulations, un nouveau fichier, portant le nom de la classe suivi de l'extension .cs, est disponible dans la hiérarchie à gauche. Par défaut, ce fichier sera ouvert et contient la classe qui a été déclarée dans l'espace de noms correspondant au dossier de destination.

1.1.2 L'héritage

Il existe un concept extrêmement important en POO : l'héritage. Globalement, si vous avez la possibilité de dire « X est un Y », l'équivalent pourrait être de dire « X hérite de Y ». X reprend toutes les propriétés et tous les comportements de Y, mais le spécifie. Donnons un exemple concret : « Un Mac est un ordinateur ». Donc, au niveau du développement orienté objet, un Mac reprend toutes les propriétés d'un ordinateur ainsi que ses comportements, mais les spécifie en y apportant ses propres éléments. On dit dans ces cas-là que `Mac` est une classe fille de la classe `Ordinateur`.

En C#, cette notion est centrale car tous les éléments que vous allez manipuler héritent naturellement de la classe `System.Object`, qui définit le comportement de base de n'importe quel objet. De surcroît, contrairement à d'autres langages (comme le C++), il n'est pas possible, en C#, d'hériter de plusieurs classes : une seule classe mère est possible. Si aucune classe mère n'est spécifiée, c'est par définition la classe `System.Object` qui constitue la classe mère (sans qu'une quelconque manipulation soit requise).

Remarque

En C#, il n'est pas possible d'hériter de plusieurs classes. Il faut donc choisir la classe dont on hérite. En l'absence de précision, le compilateur génère automatiquement, de façon transparente, un héritage de la classe `System.Object`, comme décrit ci-dessus. Si nous spécifions un héritage, cela ne veut pas dire que la classe hérite de `System.Object` ET de la classe héritée, mais uniquement de la classe héritée, qui remplace l'héritage généré par le compilateur. La classe héritée, elle-même, hérite soit d'une autre classe, soit directement de `System.Object`. En finalité, toutes les classes en C# héritent d'une façon ou d'une autre de `System.Object`.

Afin d'indiquer qu'une classe hérite d'une autre, il faut utiliser le deux-points, suivi de la classe dont on souhaite hériter :

```
class Ordinateur { }
class Mac : Ordinateur { }
```

Bien entendu, ce n'est pas parce qu'une classe hérite d'une autre qu'elle a forcément accès à tout ce qui a été défini au sein de la classe mère.

1.1.3 L'encapsulation

Tout ce qui se trouve à l'intérieur d'une classe est désigné par un terme bien spécifique : l'encapsulation. Avec celle-ci vient également la notion de portée, qui indique comment les choses sont perçues d'un point de vue extérieur à la classe.

La portée permet de définir la visibilité d'un élément d'une classe ou de la classe elle-même. Il existe en tout sept portées en C# :

- `public` : définit que l'élément est totalement visible dans et en dehors de la classe.
- `private` : définit que l'élément n'est visible qu'au sein de la classe où il est déclaré alors qu'il est totalement invisible de l'extérieur.
- `internal` : définit que l'élément est visible uniquement au sein du projet où il est déclaré. Nous pouvons considérer l'élément comme étant public, mais simplement au sein du projet dans lequel il est déclaré. Un autre projet qui référence notre projet n'a pas connaissance d'un élément déclaré comme `internal`. Par défaut, en l'absence de portée explicite sur une classe, c'est la portée `internal` qui est sélectionnée par le compilateur.
- `protected` : définit que l'élément est visible uniquement au sein de la classe où il est déclaré ainsi que dans sa hiérarchie de classes filles. Cela rejoint le concept de l'héritage, que nous verrons plus loin dans ce chapitre.
- `protected internal` : définit un cumul entre `protected` et `internal`. Un élément déclaré avec cette portée est visible par la classe concernée ainsi que ses classes filles, tout comme par toutes les autres classes au sein du même projet. Cela signifie également que si une classe fille est déclarée en dehors du projet actuel, elle peut accéder à un élément `protected internal`, tout comme n'importe quelle classe du même projet.
- `private protected` : définit une intersection entre `protected` et `internal`. Un élément déclaré avec cette portée n'est visible que par la classe concernée ainsi que les classes filles qui sont définies au sein du même projet. Cela veut dire qu'une classe fille définie en dehors du projet actuel ne pourra pas accéder à cet élément.

– `file` : ajoutée en C# 11, cette portée définit une visibilité uniquement dans le cadre du fichier en cours. Cette portée est très particulière car elle n'a pas vocation à être directement utilisée par les développeurs. Elle existe surtout pour les outils de génération automatique de code. Néanmoins, dans de très rares cas, il peut être utile de déclarer un élément qui n'existe que dans le cadre d'un fichier pour un algorithme précis. Il est à noter également que, contrairement aux autres portées, cette portée n'est valide que pour la déclaration d'un type. Elle ne peut pas s'appliquer sur une méthode, un champ ou une propriété.

Avec toutes ces portées, il est possible de créer la classe qui correspond finement au besoin de votre application, pour éviter que certains éléments ne sortent du périmètre de la classe. En reprenant notre exemple, considérons que la classe `Ordinateur` dispose d'un booléen indiquant si la machine est allumée ou non. Afin d'éviter que quelqu'un ne puisse manipuler directement cette donnée, la manière de procéder est de la définir comme étant publiquement accessible en lecture, mais privée pour ce qui est de l'écriture. En conséquence, seule une méthode publique, définie dans cette classe, comme par exemple `Allumer` ou `Eteindre`, peut changer la valeur de cet indicateur. Nous nous préservons ainsi d'un changement d'état non maîtrisé (car nous pouvons considérer que l'opération d'extinction nécessite d'effectuer quelques opérations en amont avant de basculer le booléen).

1.2 Que peut-on déclarer dans une classe ?

Nous l'avons vu, il existe deux types d'éléments que nous pouvons déclarer dans une classe : des méthodes (actions) et des données. Voyons rapidement comment les déclarer.

1.2.1 Les méthodes

Une méthode traduit une action qu'il est possible d'invoquer sur la classe. Lors de la déclaration d'une méthode, il faut se poser les questions suivantes :

– S'agit-il d'une action qui doit pouvoir être réalisée depuis l'extérieur ou uniquement depuis l'intérieur de la classe ?

– Est-ce qu'une valeur de retour particulière est attendue ?

– Certaines informations sont-elles nécessaires pour que cette méthode fonctionne ?

Vous avez déjà eu un aperçu d'un appel de méthode dans le premier chapitre, sur la classe `Console` : `WriteLine` et `ReadLine`. Ces deux méthodes illustrent bien les points cités précédemment :

– `WriteLine` doit pouvoir être appelée depuis l'extérieur. Nous n'attendons pas de valeur en retour à son appel, mais il est nécessaire de lui transmettre l'information que nous souhaitons écrire.
– `ReadLine` doit également pouvoir être appelée depuis l'extérieur. Nous avons besoin de récupérer l'information saisie par l'utilisateur uniquement, sans besoin de lui transmettre une quelconque information.

La syntaxe de déclaration d'une méthode dans une classe est la suivante :

```
PORTÉE [static] TYPE_RETOUR NOM_METHODE([PARAMÈTRES])
```

Le type de retour doit correspondre à un type C# connu. Par exemple, si nous souhaitons créer une méthode qui réalise l'addition de deux nombres et renvoie le résultat, le tout accessible publiquement, nous la déclarons comme suit :

```
public int Addition(int premier, int second) {}
```

Remarque

Dès lors que nous déclarons une méthode avec une valeur de retour sans écrire le contenu de la méthode, le compilateur émet immédiatement une erreur de compilation. Ceci est dû au fait que chaque méthode retournant un résultat doit obligatoirement comporter une instruction `return`.

Lorsqu'une méthode doit renvoyer une valeur, il faut utiliser le mot-clé `return` afin de définir la valeur que nous souhaitons renvoyer. L'instruction `return` peut être utilisée directement avec une valeur ou alors nous pouvons nous servir d'une variable du type de retour attendu. Dans le cas de l'exemple ci-dessus, ces deux façons d'écrire la méthode sont valides :

```
public int Addition(int premier, int second)
{
    return premier + second;
}
```

```
public int Addition(int premier, int second)
{
    int resultat = premier + second;
    return resultat;
}
```

Un élément important à garder en mémoire : à l'instar de ce que nous avons vu dans le chapitre précédent avec la déclaration de classes du même nom au sein du même espace de noms, il n'est pas possible de déclarer deux fois la même méthode à l'intérieur d'une même classe. Si les noms sont identiques et que les paramètres le sont également, alors le compilateur C# considère qu'il s'agit de la même méthode. La valeur de retour ne constitue pas un élément distinctif. Ainsi, la déclaration des deux méthodes suivantes dans la même classe est impossible et cela provoque une erreur de compilation :

```
public int Addition (int premier, int second)
{
    return premier + second;
}
public void Addition (int premier, int second)
{
}
```

Remarque

Comme nous pouvons le constater dans l'exemple ci-dessus, le mot-clé void précise que la méthode ne renvoie aucun résultat. La notion de type de retour étant obligatoire, il faut utiliser ce mot-clé pour indiquer les cas où il n'y en a pas.

Si la méthode ne prend pas de paramètres, la présence de parenthèses ouvrantes et fermantes accolées au nom de la méthode est malgré tout nécessaire pour signifier qu'il s'agit d'une méthode :

```
public void MaMethode()
{
}
```

Au sein d'une méthode qui déclare son propre bloc, il est possible de déclarer des variables et constantes qui sont considérées uniquement comme locales (c'est-à-dire visibles au sein de la méthode et de tous ses sous-blocs, mais invisibles dans les blocs parents, directs ou indirects).

Chapitre 3
Profilage d'une application .NET

1. Gestion de la mémoire par .NET

1.1 Principes de base

La plate-forme .NET comprend un gestionnaire de mémoire en charge de l'affectation et de la libération de la mémoire. Ces deux opérations fonctionnent bien sûr de concert, mais dans un mode très différent de C++ ou d'autres langages où la mémoire est gérée manuellement par le développeur.

En .NET, au lieu de laisser au développeur le soin de gérer ces deux opérations, le runtime prend en charge la totalité de la seconde opération, à savoir le nettoyage de la mémoire. Le développeur réserve de la mémoire en créant une instance d'une classe grâce au mot-clé **new**, mais ne se trouve pas dans l'obligation de la libérer explicitement par la suite (même s'il reste possible de mettre en place des mécanismes avancés de nettoyage de la mémoire). C'est le rôle d'un module de .NET appelé ramasse-miettes (ou *Garbage Collector*, parfois abrégé en GC, en anglais). Ce dernier se chargera, lorsque le besoin s'en fait sentir, de collecter la mémoire désormais inutile et de la remettre à disposition du programme. Nous allons détailler son fonctionnement dans les chapitres suivants.

La majorité des développeurs .NET ne prennent pas garde à la façon dont la mémoire est gérée. Il s'agit ici d'un bel accomplissement de la plate-forme, car si cela est autant transparent, c'est que le processus est efficace. Néanmoins, connaître le fonctionnement de cette gestion permet, dans les cas les plus avancés, de pouvoir répondre et traiter d'éventuels problèmes, que nous détaillerons plus en détail au fil du chapitre.

1.2 Gestion de mémoire automatisée et performances

Comme pour toutes les techniques simplifiant le travail de programmation, des polémiques sont nées sur la perte de performance liée à l'utilisation des ramasse-miettes. Pour couper court à une discussion potentiellement stérile, revenons simplement aux principes d'amélioration de la performance d'une application, tels qu'exposés au début du présent ouvrage. Effectivement, gérer la mémoire manuellement peut permettre de gagner quelques pourcents sur le temps passé à l'exécution dans une fonctionnalité. Mais la mise en œuvre de la gestion manuelle est complexe : même des développeurs expérimentés font des erreurs de gestion de mémoire en C++, et cette gestion peut représenter une part non négligeable du temps affecté au développement.

En résumé, le ratio « temps de codage/performance gagnée » est très mauvais, et dans la majorité des cas, nous aurons avantage à utiliser une technologie de collecte automatisée. D'autant plus que les algorithmes de nettoyage de la mémoire ont été améliorés au fil du temps et disposent de modes d'exécution différents selon l'environnement de destination, ce qui laisse tout de même une petite place à la configuration et l'optimisation des performances à ce niveau.

1.3 Le cas particulier du temps réel

1.3.1 Lever un malentendu

Il existe un champ d'application où les ramasse-miettes posent de réels problèmes, à savoir les applications temps réel. Il est important d'en parler, car la confusion entre les notions de rapidité et de temps réel a amplement participé au mythe des mauvaises performances des langages à gestion de mémoire automatique.

La programmation dite temps réel consiste à proposer des fonctions dont le temps d'exécution est déterministe, c'est-à-dire qu'au moment où le développeur crée une fonctionnalité en appelant des fonctions du langage, il peut calculer à l'avance la durée d'exécution de celle-ci, en additionnant les temps unitaires connus. La notion de temps réel s'arrête là et n'englobe aucune notion de rapidité dans l'exécution.

Remarque

La programmation temps réel est très liée à la théorie de la gestion des risques. Un risque, en termes mathématiques, est décrit par l'équation Risque = Probabilité x Gravité. Appliquons ceci sur le cas d'un logiciel qui doit réagir dans un temps imparti, en supposant qu'il y a une chance sur mille qu'au cours d'une journée d'utilisation, ce ne soit pas le cas (c'est la probabilité). Si la gravité est minime, par exemple, simplement faire attendre une demi-seconde un opérateur, le risque est acceptable. Si à l'inverse la vie de centaines de personnes est en jeu, il est évident que le risque est alors beaucoup trop élevé.

Encore une fois, c'est un malentendu qui est à l'origine de l'association entre "temps réel" et "rapidité d'exécution". Le temps réel est utilisé dans l'aéronautique et l'industrie des satellites, mais pas pour des raisons de performance. C'est d'ailleurs un motif de surprise pour beaucoup de gens de découvrir que les processeurs industriels sont cadencés bien moins haut que les processeurs grand public. Dans un ordinateur portable, le processeur effectue plusieurs milliards d'opérations par seconde (notion de gigahertz), mais au prix d'un système de refroidissement nécessitant l'apport d'air frais extérieur, voire plus (comme du watercooling, ou pour les cas les plus extrêmes, un refroidissement à l'azote). En effet, l'enveloppe thermique, c'est-à-dire la chaleur diffusée par le processeur, augmente exponentiellement avec la fréquence d'opération. Dans un environnement industriel, où des robots sont exposés à la poussière et aux projections, il est évidemment hors de question que le processeur ne soit pas abrité par une protection étanche. Du coup, ces processeurs fonctionnent à une cadence exprimée en millions d'opérations par seconde (ou mégahertz) « seulement », de façon à dégager moins de chaleur et ainsi pouvoir opérer enfermés dans leur coque de protection.

Le risque de surchauffe est bien sûr associé à la panne ; les processeurs modernes sont capables de mettre automatiquement le système en veille, ou au minimum de se couper, avant que la chaleur ne les endommage de manière définitive. Il est donc aisé de comprendre que ce genre de comportement n'est pas tolérable pour certains cas d'usage, comme dans un avion par exemple.

Les processeurs utilisés ne présentent évidemment pas ce genre de comportement. Vu leur fréquence d'utilisation, une surchauffe ne peut provenir que d'un évènement extérieur grave, comme un incendie à bord. Et dans ce cas gravissime, il vaut bien sûr mieux que le processeur (et donc certaines fonctions vitales de l'avion) continue de fonctionner plutôt que de s'arrêter, ce qui ne l'empêchera de toute façon pas de brûler.

Nous touchons là au cœur de la problématique des applications temps réel : quel que soit le contexte, elles doivent continuer à fonctionner de manière déterministe, comme prévu. Mais à nouveau, la performance n'est pas le problème. Un avion ne sera pas plus sûr avec un processeur opérant à 1 GHz qu'avec un processeur opérant à 1 MHz, bien au contraire. En termes de rapidité de réaction, en supposant par exemple que l'envoi d'une commande de vol nécessite quelques dizaines d'opérations, la différence entre les deux systèmes ne sera que de quelques millionièmes de seconde. Ce delta est négligeable. En revanche, même s'il n'y a qu'une chance sur un million que le premier processeur se mette en sécurité parce que, par exemple, le soleil tape un peu plus fort que d'habitude, voudrions-nous monter dans un avion qui en est équipé, sachant qu'un avion moderne vole autour de 100 000 heures sur l'ensemble de sa vie ? Certainement pas...

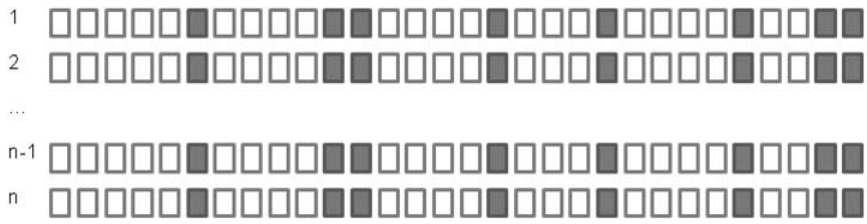
1.3.2 Non-déterminisme des ramasse-miettes

Voilà pourquoi la technologie du ramasse-miettes est incompatible avec les applications temps réel et que nous ne verrons jamais une étiquette "Powered by .NET" sur les commandes de vol d'un avion.

Comme la gestion de la mémoire est prise en charge par le système et non le développeur, il est impossible de savoir à l'avance quand le ramasse-miettes va être déclenché. Le runtime peut choisir de le déclencher parce que le système d'exploitation lui signale qu'il a besoin de mémoire pour une autre application, ou parce qu'il considère grâce à ses propres algorithmes que la mémoire nécessite un nettoyage.

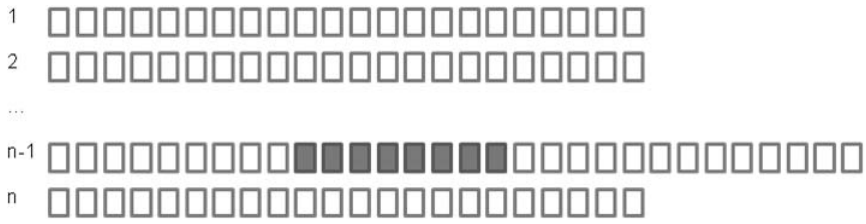
Lorsqu'un développeur utilise un langage où il possède la maîtrise de l'affectation et du nettoyage de la mémoire (comme le C ou le C++), il peut agir de façon régulière, de telle sorte que les opérations d'affectation et de nettoyage se produisent à des moments prévisibles. De la sorte, à chaque lancement de l'application ou à chaque exécution d'un traitement donné, la position de ces opérations ne change pas.

Pour illustrer cela par une image, on peut considérer que chaque rectangle plein constitue une opération de nettoyage de la mémoire, alors que chaque rectangle blanc constitue une lecture ou une affectation de la mémoire :



À chaque lancement de l'application ou du traitement, la position de ces opérations ne change pas.

À l'inverse, le schéma ci-dessous montre le même traitement codé dans un environnement de développement où le moteur d'exécution est désormais responsable du recyclage mémoire. Les rectangles pleins correspondent à l'activation du ramasse-miettes.



Le recyclage de la mémoire est alors aléatoire, le ramasse-miettes se déclenchant au bon vouloir du moteur d'exécution. Pendant certains traitements, il ne se déclenche jamais. Sur d'autres, il est actif à la fin du traitement, voire au début ou même en plein milieu s'il le faut.

Le mécanisme de ramasse-miettes est bien plus complexe qu'il n'y paraît. Bien qu'il ne soit pas complètement prédictible, il cherche toujours à s'exécuter de la meilleure manière possible : en attendant au maximum qu'une interaction utilisateur soit terminée, ou que le processeur ne soit plus trop sollicité. Il va même jusqu'à se lancer dans un thread parallèle à celui du traitement courant de l'application pour la gêner le moins possible. Enfin, des mécanismes sophistiqués, que nous détaillons plus loin, lui permettent de faire son travail le plus vite possible, en ne nettoyant que la partie de la mémoire qui a le plus de chance de contenir des espaces recyclables.

Malgré tout cela, il reste un risque minime que le ramasse-miettes se déclenche en plein milieu d'une utilisation du logiciel et provoque un ralentissement notable. L'utilisateur d'une application de gestion ne se rend, la plupart du temps, même pas compte que le serveur a pris une seconde de plus. À l'inverse, il est impossible de prendre le risque, même ténu, qu'un avion ne réagisse plus aux commandes de vol pendant cette même seconde.

Clairement, les applications temps réel sont extrêmement différentes des logiciels standards. Le niveau de test et de validation qui leur est appliqué est bien évidemment beaucoup plus poussé, les plates-formes de développement et/les matériels utilisés sont spécifiques, et surtout leur développement est énormément plus long et coûteux.

Pour revenir aux applications habituelles (c'est-à-dire ne nécessitant pas de temps réel), le lecteur peut être assuré que la perte de performance liée à une gestion automatique de la mémoire est un mythe. Les ralentissements effectifs sont presque toujours invisibles à l'utilisateur. Mais surtout, l'utilisation de cette technique réduit très fortement les fuites de mémoire, alors que seuls un excellent niveau de développement ou une expertise supplémentaire peuvent nous garantir la même chose en C++, au prix dans un cas comme dans l'autre d'un important surcoût.

Remarque

L'analyse du graphique ci-dessus peut amener à la remarque suivante : vu que l'application ne gère normalement pas le recyclage mémoire pendant l'exécution d'une fonction, serait-il possible qu'une application .NET soit alors plus rapide qu'une application C++ ? C'est effectivement le cas : des benchmarks ont montré que des opérations limitées dans le temps et très consommatrices de mémoire pouvaient être traitées ponctuellement plus rapidement en .NET qu'en C++. Et si l'application a des plages de moindre utilisation que le ramasse-miettes peut utiliser, le même résultat peut être atteint de manière globale. À l'inverse, il sera en faveur de C++ si l'application est utilisée de manière continuellement intensive, à condition bien sûr que le codage soit d'une qualité parfaite en termes de gestion des fuites mémoire.

1.4 Affectation de la mémoire

Cette justification du recyclage automatique de la mémoire étant posée, nous allons dans un premier temps nous intéresser à l'opération inverse, à savoir l'affectation de la mémoire. Comment un développeur C# demande-t-il de la mémoire à .NET ?

Le mot-clé **`new`**, bien que son fonctionnement interne soit relativement complexe comme nous allons le voir ci-dessous, est extrêmement simple à utiliser. Considérons deux classes **`Animal`** et **`Chien`**, la seconde héritant de la première :