

IHM, Winforms, exceptions, flux



-
- **Programmation événementielle**
 - **Les événements**
 - **Propriétés et indexeurs**
 - **Fenêtres et ressources**
 - **Contrôles dans les formulaires**
 - **Exceptions comparées**
 - **Flux et fichiers : données simples**

Programmation événementielle et visuelle



Plan du chapitre: 

Introduction

1. Programmation visuelle basée sur les pictogrammes

2. Programmation orientée événements

3. Normalisation du graphe événementiel

le graphe événementiel arcs et sommets
les diagrammes d'états UML réduits

4. Tableau des actions événementielles

5. Interfaces liées à un graphe événementiel

6. Avantages et modèle de développement RAD visuel

le modèle de la spirale (B.Boehm)
le modèle incrémental

1. Programmation visuelle basée sur les pictogrammes

Le développement visuel rapide d'application est fondé sur le concept de programmation visuelle associée à la montée en puissance de l'utilisation des Interactions Homme-Machine (IHM) dont le dynamisme récent ne peut pas être méconnu surtout par le débutant. En informatique les systèmes MacOS, Windows, les navigateurs Web, sont les principaux acteurs de l'ingénierie de l'IHM. Actuellement dans le développement d'un logiciel, un temps très important est consacré à l'ergonomie et la communication, cette part ne pourra que grandir dans un avenir proche; car les utilisateurs veulent s'adresser à des logiciels efficaces (ce qui va de soi) mais aussi conviviaux et faciles d'accès.

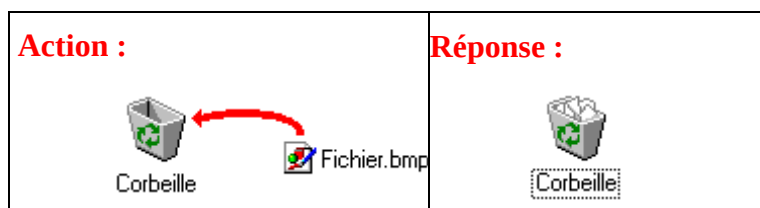
Les développeurs ont donc besoin d'avoir à leur disposition des produits de développement adaptés aux nécessités du moment. A ce jour la programmation visuelle est une des réponses à cette attente des développeurs.

La programmation visuelle au tout début a été conçue pour des personnes n'étant pas des programmeurs en basant ses outils sur des manipulations de pictogrammes.

Le raisonnement communément admis est qu'un dessin associé à une action élémentaire est plus porteur de sens qu'une phrase de texte.

A titre d'exemple ci-dessous l'on enlève le "fichier.bmp" afin de l'effacer selon deux modes de communication avec la machine: utilisation d'icônes ou entrée d'une commande textuelle.

Effacement avec un langage d'action visuelle (souris)



Effacement avec un langage textuel (clavier)

Action : <code>del c:\Exemple\Fichier.bmp</code>	Réponse : ?
--	-------------------------

Nous remarquons donc déjà que l'interface de communication MacOS, Windows dénommée "bureau électronique" est en fait un outil de programmation de commandes systèmes.

Un langage de programmation visuelle permet "d'écrire" la partie communication d'un programme uniquement avec des dessins, diagrammes, icônes etc... Nous nous intéressons aux systèmes RAD (Rapid Application Development) visuels, qui sont fondés sur des langages objets à bases d'icônes ou pictogrammes. Visual Basic de MicroSoft est le premier RAD visuel à avoir été commercialisé dès 1991, il est fondé sur un langage Basic étendu incluant des objets étendus en VB.Net depuis 2001, puis dès 1995 Delphi le premier RAD visuel de Borland fondé sur Pascal objet, puis actuellement toujours de Borland : C++Builder RAD visuel fondé sur le langage C++ et Jbuilder, NetBeans RAD visuel de Sun fondés sur le langage Java, Visual C++, Visual J++ de Microsoft, puis Visual studio de Microsoft etc...

Le développeur trouve actuellement, une offre importante en outil de développement de RAD visuel y compris en open source. Nous proposons de définir un langage de RAD visuel ainsi :

Un **langage visuel** dans un RAD visuel est un **générateur de code source** du langage de base qui, derrière chaque action visuelle (dépôt de contrôle, click de souris, modifications des propriétés, etc...) engendre des lignes de code automatiquement et d'un manière transparente au développeur.

Des outils de développement tels que Visual Basic, Delphi ou C# sont adaptés depuis leur création à la programmation visuelle pour débutant. Toutefois l'efficacité des dernières versions a étendu leur champ au développement en général et dans l'activité industrielle et commerciale avec des versions "entreprise" pour VB et "Architect" pour Delphi et les versions .Net.

En outre, le système windows est le plus largement répandu sur les machines grand public (90% des PC vendus en sont équipés), il est donc très utile que le débutant en programmation sache utiliser un produit de développement (rapide si possible) sur ce système.

Proposition :

Nous considérons dans cet ouvrage, la programmation visuelle à la fois comme une **fin** et comme un **moyen**.

La programmation visuelle est sous-tendue par la réactivité des programmes en réponse aux actions de l'utilisateur. Il est donc nécessaire de construire des programmes qui répondent à des

sollicitations externes ou internes et non plus de programmer séquentiellement (ceci est essentiellement dû aux architectures de Von Neumann des machines) : ces sollicitations sont appelées des événements.

Le concept de **programmation dirigée ou orientée par les événements** est donc la composante essentielle de la programmation visuelle.

Terminons cette présentation par 5 remarques sur le concept de RAD :

Utiliser un RAD simple mais nuisant

Nous ne considérerons pas comme utile pour des débutants de démarrer la programmation visuelle avec des RAD basés sur le **langage C++**. Du fait de sa large permissivité ce langage permet au programmeur d'adopter certaines **attitudes dangereuses** sans contrôle possible. Seul le programmeur confirmé au courant des pièges et des subtilités de la programmation et du langage, pourra exploiter sans risque la richesse de ce type de RAD.

Avoir de bonnes méthodes dès le début

Le RAD Visual C# inclus dans Visual Studio, a des **caractéristiques très proches de celles de ses parents Java et Delphi**, tout en apportant quelques améliorations. L'aspect fortement typé du langage C# autorise la prise en compte par le développeur débutant de bonnes attitudes de programmation.

C'est Apple qui en a été le promoteur

Le premier environnement de développement visuel professionnel fut basé sur **Object Pascal** et a été conçu par Apple pour le système d'exploitation **MacOs**, sous la dénomination de **MacApp** en 1986. Cet environnement objet visuel permettait de développer des applications MacIntosh avec souris, fenêtre, menus déroulants etc...

Microsoft l'a popularisé

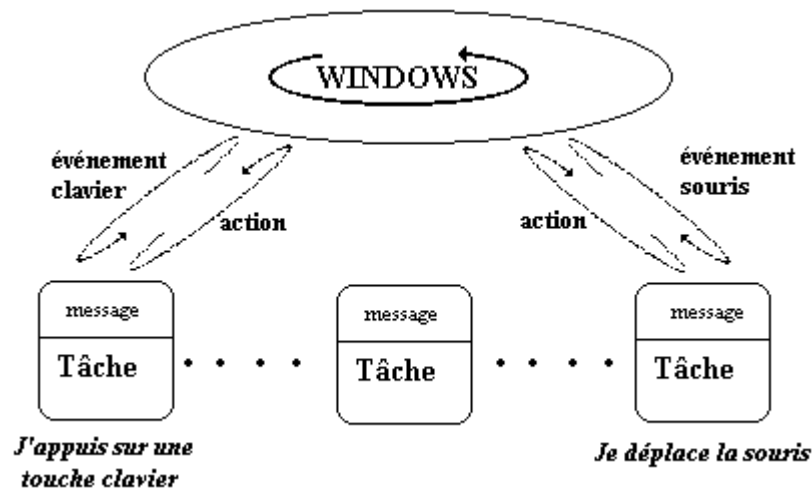
Le RAD Visual Basic de MicroSof conçu à partir de 1992, basé sur le langage Basic avait pour objectif le développement de petits logiciels sous Windows par des **programmeurs non expérimentés et occasionnels**. Actuellement il se décline en VB.Net un langage totalement orienté objet faisant partie intégrante de la plate-forme .Net Framework de Microsoft avec Visual C#.

Le concept de RAD a de beaux jours devant lui

Le métier de développeur devrait à terme, consister grâce à des outils tels que les RAD visuels, à prendre un "caddie" et à aller dans un supermarché de composants logiciels génériques adaptés à son problème. Il ne lui resterait plus qu'à assembler le flot des événements reliant entre eux ces logiciels en kit.

2. Programmation orientée événements

Sous les versions actuelles de Windows, système multi-tâches préemptif sur micro-ordinateur, les concepts quant à la programmation par événement restent sensiblement les mêmes que sous les anciennes versions.



Nous dirons que le système d'exploitation passe l'essentiel de son " temps " à attendre une action de l'utilisateur (événement). Cette action déclenche un message que le système traite et envoie éventuellement à une application donnée.

Une définition de la programmation orientée événements

Logique de conception selon laquelle un programme est construit avec des objets et leurs propriétés et d'après laquelle les interventions de l'utilisateur sur les objets du programme déclenchent l'exécution des routines associées.

Par la suite, nous allons voir dans ce chapitre que la programmation d'une application " windows-like " est essentiellement une **programmation par événements associée à une programmation classique**.

Nous pourrions construire un logiciel qui réagira sur les interventions de l'utilisateur si nous arrivons à intercepter dans notre application les messages que le système envoie. Or l'environnement RAD (C#, comme d'ailleurs avant lui Visual Basic de Microsoft), autorise la consultation de tels messages d'un façon simple et souple.

Deux approches pour construire un programme

- ❑ L'**approche événementielle** intervient principalement dans l'interface entre le logiciel et l'utilisateur, mais aussi dans la liaison dynamique du logiciel

avec le système, et enfin dans la sécurité.

- ❑ L'**approche visuelle** nous aide et simplifie notre tâche dans la construction du dialogue homme-machine.
- ❑ La combinaison de ces deux approches produit un logiciel habillé et adapté au système d'exploitation.

Il est possible de relier certains objets entre eux par des relations événementielles. Nous les représenterons par un graphe (structure classique utilisée pour représenter des relations).

Lorsque l'on utilise un système multi-fenêtré du genre windows, l'on dispose du clavier et de la souris pour agir sur le système. En utilisant un RAD visuel, il est possible de **construire un logiciel qui se comporte comme le système sur lequel il s'exécute**. L'intérêt est que l'utilisateur aura moins d'efforts à accomplir pour se servir du programme puisqu'il aura des fonctionnalités semblables au système. Le fait que l'utilisateur reste dans un **environnement familier** au niveau de la manipulation et du confort du dialogue, assure le logiciel d'un capital confiance de départ non négligeable.

3. Normalisation du graphe événementiel

Il n'existe que peu d'éléments accessibles aux débutants sur la programmation orientée objet par événements. Nous construisons une démarche méthodique pour le débutant, en partant de remarques simples que nous décrivons sous forme de schémas dérivés des diagrammes d'états d'UML. Ces schémas seront utiles pour nous aider à décrire et à implanter des relations événementielles en C# ou dans un autre RAD événementiel.

Voici deux principes qui pour l'instant seront suffisants à nos activités de programmation.

Dans une interface windows-like nous savons que:

- ❑ Certains événements déclenchent immédiatement des actions comme par exemple des appels de routines système.
- ❑ D'autres événements ne déclenchent pas d'actions apparentes mais activent ou désactivent certains autres événements système.

Nous allons utiliser ces deux principes pour conduire notre programmation par événements.

Nous commencerons par **le concept d'activation et de désactivation**, les autres événements fonctionneront selon les mêmes bases. Dans un enseignement sur la programmation événementielle, nous avons constaté que ce **concept était suffisant** pour que les étudiants comprennent les fondamentaux de l'approche événementielle.

Remarque :

Attention! Il ne s'agit que d'une manière particulière de conduire notre programmation, ce ne peut donc être ni la seule, ni la meilleure (le sens accordé au mot meilleur est relatif au domaine pédagogique). Cette démarche s'est révélée être fructueuse lors d'enseignements d'initiation à ce genre de programmation.

Hvnothèses de construction

Nous supposons donc que lorsque l'utilisateur intervient sur le programme en cours d'exécution, ce dernier réagira en première analyse de deux manières possibles :

- ❑ soit il lancera l'appel d'une routine (exécution d'une action, calcul, lecture de fichier, message à un autre objet comme ouverture d'une fiche etc...),
- ❑ soit il modifiera l'état d'activation d'autres objets du programme et/ou de lui-même, soit il ne se passera rien, nous dirons alors qu'il s'agit d'une modification nulle.

Ces hypothèses sont largement suffisantes pour la plupart des logiciels que nous pouvons raisonnablement espérer construire en initiation. Les concepts plus techniques de messages dépassent assez vite l'étudiant qui risque de replonger dans de " la grande bidouille ".

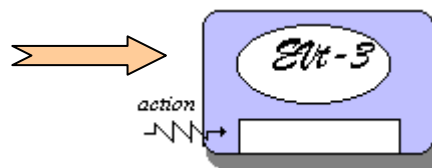
3.1 le graphe événementiel arcs et sommets

Nous proposons de construire un graphe dans lequel :

chaque **sommet** est un objet sensible à un événement donné.

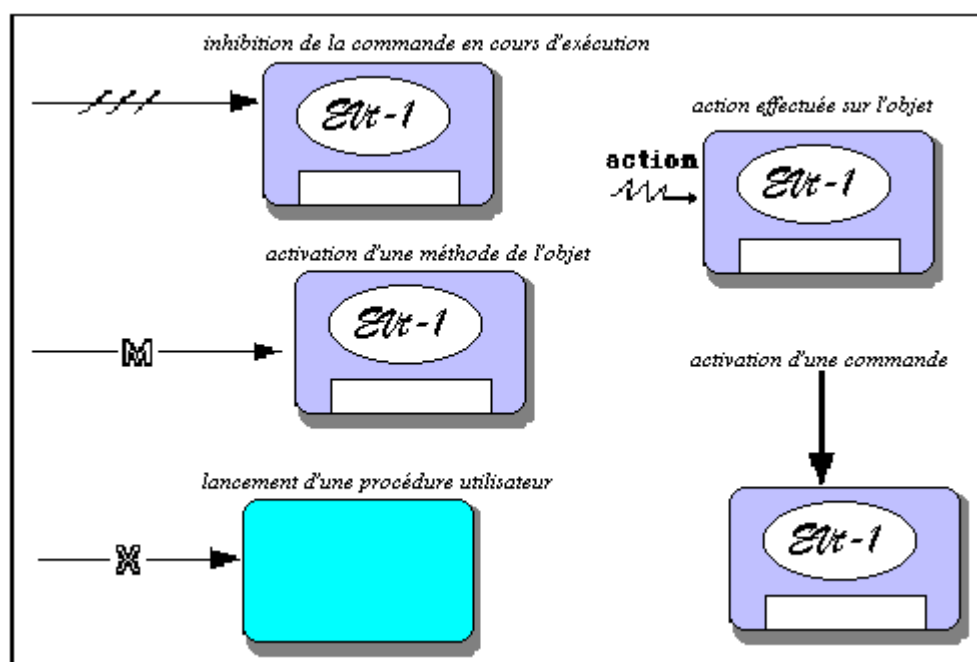


L'événement donné est déclenché par une action extérieure à l'objet.

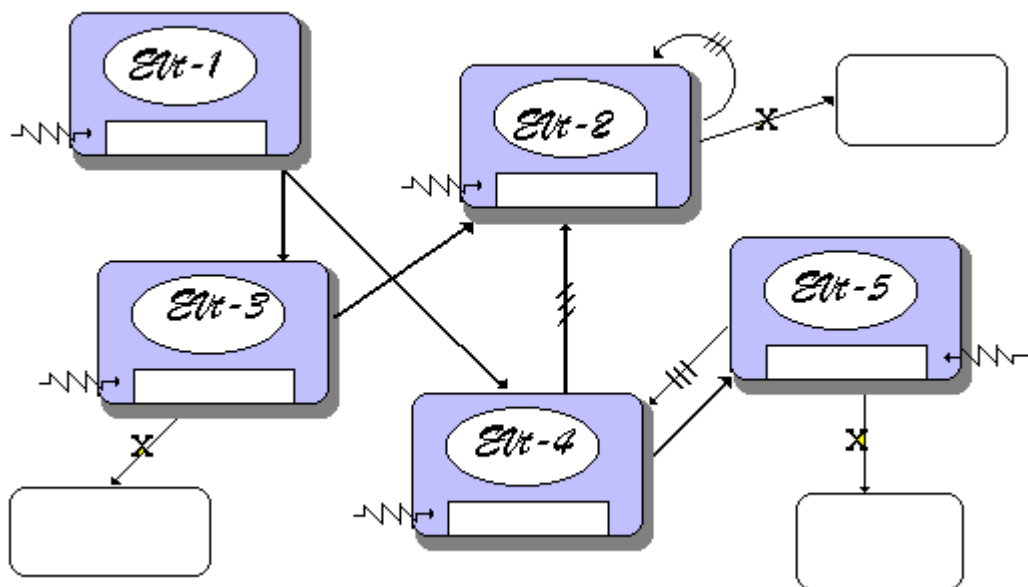


Les arcs du graphe représentent des actions lancées par un sommet.

Les actions sont de 4 types

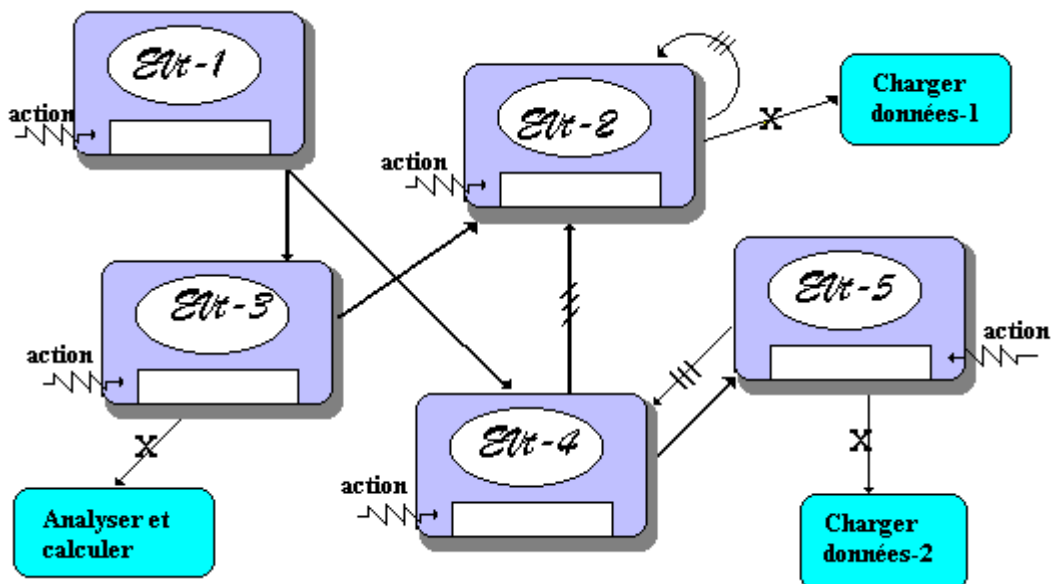


Soit le graphe événementiel suivant composé de 5 objets sensibles chacun à un événement particulier dénoté Evt-1,..., Evt-5; ce graphe comporte des réactions de chaque objet à l'événement auquel il est sensible :



Imaginons que ce graphe corresponde à une analyse de chargements de deux types différents de données à des fins de calcul sur leurs valeurs.

La figure suivante propose un tel graphe événementiel à partir du graphe vide précédent.



Cette notation de graphe événementiel est destinée à s'initier à la pratique de la description d'au maximum 4 types de réactions d'un objet sur la sollicitation d'un seul événement.

Remarques :

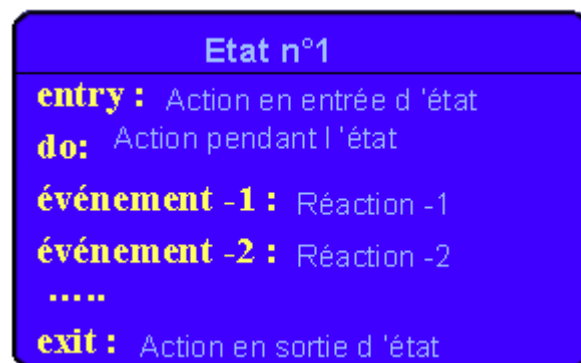
- L'arc nommé, représentant l'activation d'une méthode correspond très

exactement à la notation UML de l'envoi d'un message.

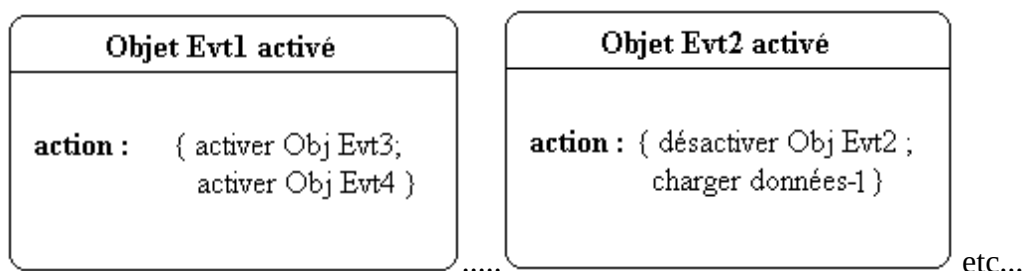
- Lorsque nous voudrions représenter d'une manière plus complète d'autres réactions d'un **seul** objet à **plusieurs événements différents**, nous pourrions utiliser la notation UML réduite de diagramme d'état pour un objet (**réduite parce qu'un objet **visuel** ne prendra pour nous, que 2 états: activé ou désactivé**).

3.2 les diagrammes d'états UML réduits

Nous livrons ci-dessous la notation générale de diagramme d'état en UML, les cas particuliers et les détails complets sont décrits dans le document de spécification d'UML.



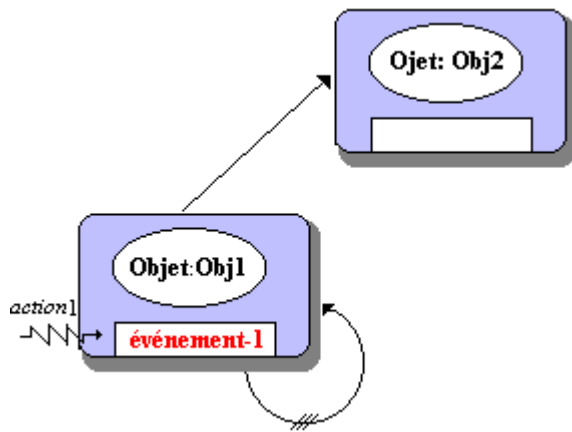
Voici les diagrammes d'états réduits extraits du graphe événementiel précédent, pour les objets **Evt-1** et **Evt-2** :



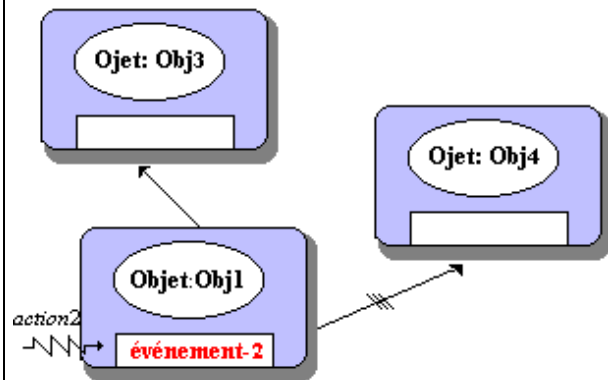
Nous remarquerons que cette écriture, pour l'instant, ne produit pas plus de sens que le graphe précédent qui comporte en sus la vision globale des interrelations entre les objets.

Ces diagrammes d'états réduits deviennent plus intéressants lorsque nous voulons exprimer le fait par exemple, qu'un seul objet **Obj1** réagit à 3 événements (événement-1, événement-2, événement-3). Dans ce cas décrivons les portions de graphe événementiel associés à chacun des événements :

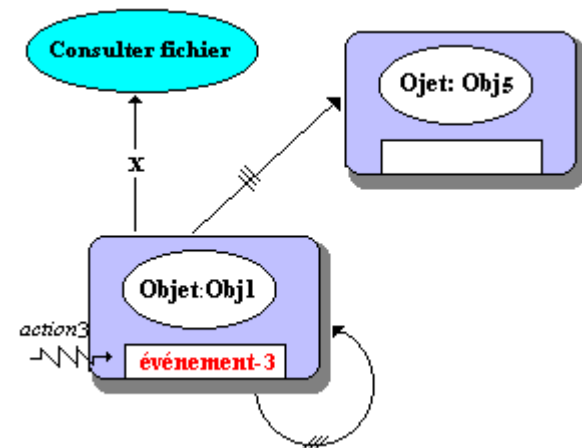
Réaction de obj1 à l'événement-1 :



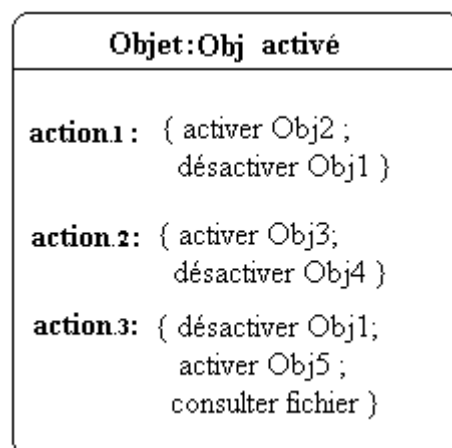
Réaction de obj1 à l'événement-2 :



Réaction de obj1 à l'événement-3 :

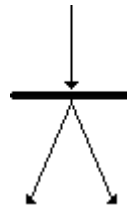
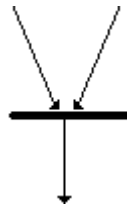


Synthétisons dans un diagramme d'état réduit les réactions à ces 3 événements :

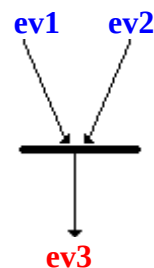


Lorsque nous jugerons nécessaire à la compréhension de relations événementielles dans un logiciel visuel, nous pourrons donc utiliser ce genre de diagramme pour renforcer la sémantique de conception des objets visuels. La notation UML sur les diagrammes d'états comprend les notions d'état de départ, de sortie, imbriqué, historisé, concurrents...

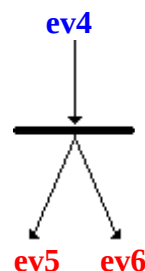
Lorsque cela sera nécessaire nous utiliserons la notation UML de synchronisation d'événements :



Dans le premier cas la notation représente la conjonction des deux événements **ev1** et **ev2** qui déclenche l'événement **ev3**.



Dans le second cas la notation représente l'événement **ev4** déclenchant conjointement les deux événements **ev5** et **ev6**.



4. Tableau des actions événementielles

L'exemple de graphe événementiel précédent correspond à une application qui serait sensible à 5 événements notés EVT-1 à EVT-5, et qui exécuterait 3 procédures utilisateur. Nous notons dans un tableau (nommé "**tableau des actions événementielles**") les résultats obtenus par analyse du graphe précédent, événement par événement..

EVT-1	EVT-3 activable EVT-4 activable
-------	------------------------------------

EVT-2	Appel de procédure utilisateur "chargement-1" désactivation de l' événement EVT-2
EVT-3	Appel de procédure utilisateur "Analyser" EVT-2 activable
EVT-4	EVT-2 désactivé EVT-5 activable
EVT-5	EVT-4 désactivé immédiatement Appel de procédure utilisateur "chargement-2"

Nous adjoignons à ce tableau une table des états des événements dès le lancement du logiciel (elle correspond à l'état initial des objets). Par exemple ici :

Evt1	activé
Evt2	désactivé
Evt3	activé
Evt4	activé
Evt5	désactivé

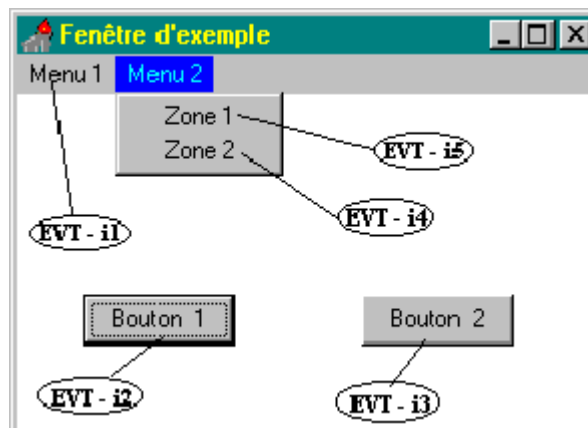
etc...

5. Interfaces liées à un graphe événementiel

construction d'interfaces liées au graphe précédent

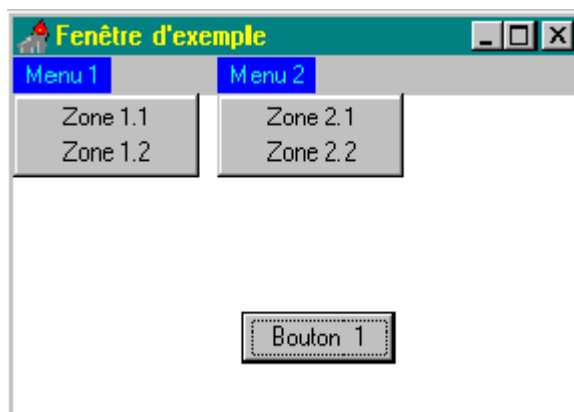
Dans l'exemple de droite, (i1,i2,i3,i4,i5)est une permutation sur (1,2,3,4,5) .

Ce qui nous donne déjà $5!=120$ interfaces possibles avec ces objets et uniquement avec cette topologie.

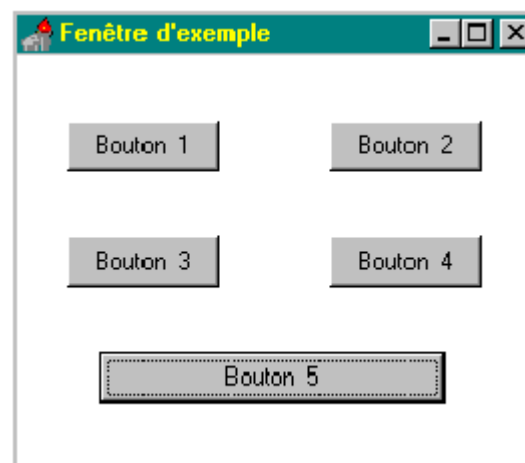


Interface n°1

La figure précédente montre une IHM à partir du graphe événementiel étudié plus haut. Ci-dessous deux autres structures d'interfaces possibles avec les mêmes objets combinés différemment et associés au même graphe événementiel :



Interface n°2



Interface n°3

Pour les choix n°2 et n°3, il y a aussi 120 interfaces possibles...

6. Avantages du modèle de développement RAD visuel

L'approche uniquement structurée (privilégiant les fonctions du logiciel) impose d'écrire du code long et compliqué en risquant de ne pas aboutir, car il faut tout tester afin d'assurer un bon fonctionnement de tous les éléments du logiciel.

L'approche événementielle préfère bâtir un logiciel fondé sur une construction graduelle en fonction des besoins de communication entre l'humain et la machine. Dans cette optique, le programmeur élabore les fonctions associées à une action de communication en privilégiant le dialogue. Ainsi les actions internes du logiciel sont subordonnées au flux du dialogue.

Avantages liés à la programmation par RAD visuel

- ❑ Il est possible de construire très rapidement un prototype.
- ❑ Les fonctionnalités de communication sont les guides principaux du développement (approche plus vivante et attrayante).
- ❑ L'étudiant est impliqué immédiatement dans le processus de conception - construction.

L'étudiant acquiert très vite comme naturelle l'attitude de réutilisation en se servant de " logiciels en kit " (soit au début des composants visuels ou non, puis par la suite ses propres composants).

Il n'y a pas de conflit ni d'incohérence avec la démarche structurée : les algorithmes étant conçus comme des boîtes noires permettant d'implanter certaines actions, seront réutilisés immédiatement.

La méthodologie objet de COO et de POO reste un facteur d'intégration général des activités du programmeur.

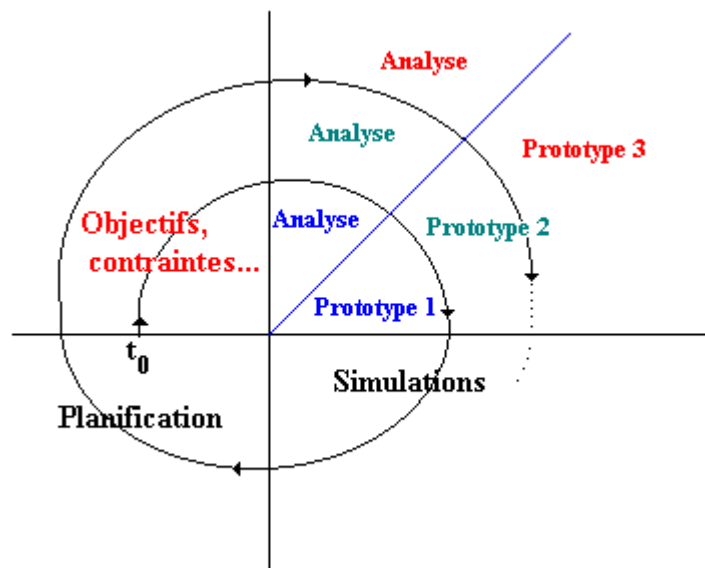
Les actions de communications classiques sont assurées immédiatement par des objets standards (composants visuels ou non) réagissant à des événements extérieurs.

Le RAD fournira des classes d'objets standards non visuels (extensibles si l'étudiant augmente sa compétence) gérant les structures de données classiques (Liste, arbre, etc..). L'extensibilité permet à l'enseignant de rajouter ses kits personnels d'objets et de les mettre à la disposition des étudiants comme des outils standards.

Modèles de développement avec un RAD visuel objet

Le développement avec ce genre de produit autorise une logique générale articulée sur la combinaison de deux modèles de développement :

le modèle de la spirale (B.Boehm) en version simplifiée

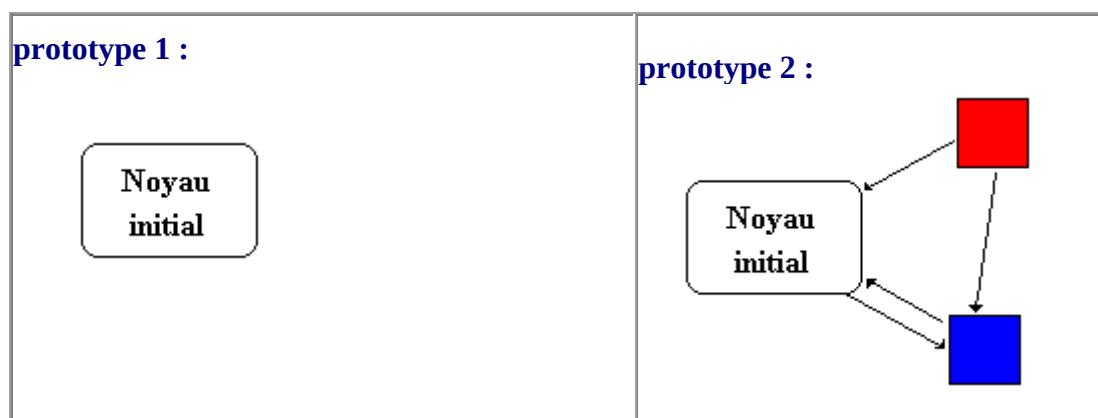


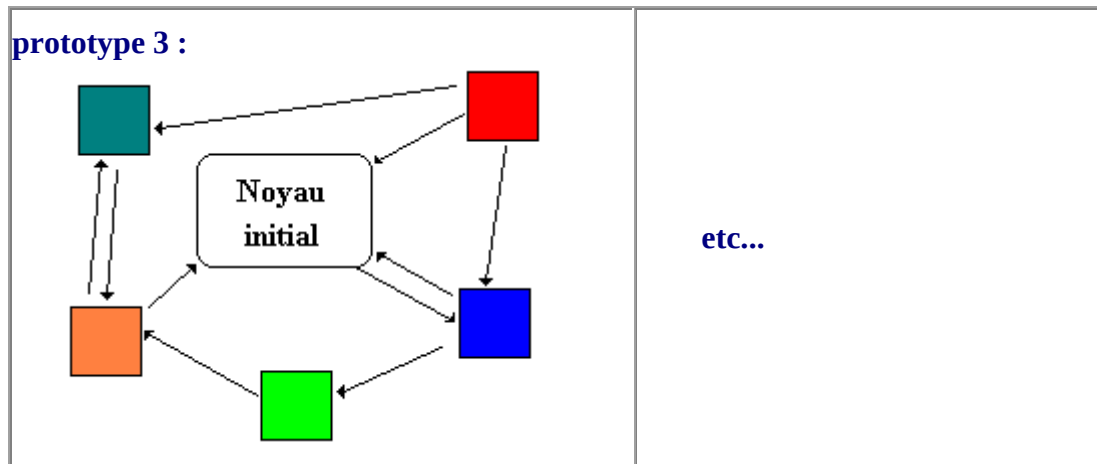
Dans le modèle de la spirale la programmation exploratoire est utilisée sous forme de prototypes simplifiés cycle après cycle. L'analyse s'améliore au cours de chaque cycle et fixe le type de développement pour ce tour de spirale.

le modèle incrémental

Il permet de réaliser chaque prototype avec un bloc central au départ, s'enrichissant à chaque phase de nouveaux composants et ainsi que de leurs interactions.

Associé à un cycle de prototypage dans la spirale, une seule famille de composants est développée pour un cycle fixé.





Ce modèle de développement à l'aide d'objets visuels ou non, fournira en fin de parcours un prototype opérationnel qui pourra s'intégrer dans un projet plus général.

Nous verrons sur des exemples comment ce type d'outil peut procurer aussi des avantages au niveau de la programmation défensive.

7. principes d'élaboration d'une IHM

Nous énumérons quelques principes utiles à l'élaboration d'une interface associée étroitement à la programmation événementielle

Notre point de vue reste celui du pédagogue et non pas du spécialiste en ergonomie ou en psychologie cognitive qui sont deux éléments essentiels dans la conception d'une interface homme-machine (**IHM**). Nous nous efforcerons d'utiliser les principes généraux des **IHM** en les mettant à la portée d'un débutant avec un double objectif :

- ❑ Faire écrire des programmes interactifs. Ce qui signifie que le programme doit communiquer avec l'utilisateur qui reste l'acteur privilégié de la communication. Les programmes sont Windows-like et nous nous servons du RAD visuel C# pour les développer. Une partie de la spécification des programmes s'effectue avec des objets graphiques représentant des classes (programmation objet visuelle).
- ❑ Le développeur peut découpler pendant la conception la programmation de son interface de la programmation des tâches internes de son logiciel (pour nous généralement ce sont des algorithmes ou des scénarios objets).

Nous nous efforçons aussi de ne proposer que des outils pratiques qui sont à notre portée et utilisables rapidement avec un système de développement RAD visuel comme C#.

Interface homme-machine, les concepts

Les spécialistes en ergonomie conceptualisent une IHM en six concepts :

- ❑ les objets d'entrée-sortie,
- ❑ les temps d'attente (temps de réponse aux sollicitations),
- ❑ le pilotage de l'utilisateur dans l'interface,
- ❑ les types d'interaction (langage, etc.),
- ❑ l'enchaînement des opérations,
- ❑ la résistance aux erreurs (ou robustesse qui est la qualité qu'un logiciel à fonctionner même dans des conditions anormales).

Un principe général provenant des psycho-linguistes guide notre programmeur dans la complexité informationnelle : la mémoire rapide d'un humain ne peut être sollicitée que par un nombre limité de concepts différents en même temps (nombre compris entre sept et neuf). Développons un peu plus chacun des six concepts composant une interface.

Les objets d'entrée-sortie

Concept

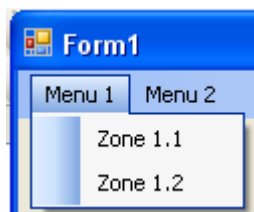
Une IHM présente à l'utilisateur un éventail d'informations qui sont de deux ordres : des commandes entraînant des actions internes sur l'IHM et des données présentées totalement ou partiellement selon l'état de l'IHM.

Les commandes participent à la " saisie de l'intention d'action" de l'utilisateur, elles sont matérialisées dans le dialogue par des **objets d'entrée** de l'information (boîtes de saisie, boutons, menus etc...).

Voici avec un RAD visuel comme C#, des objets visuels associés aux objets d'entrée de l'information, ils sont très proches visuellement des objets que l'on trouve dans d'autres RAD visuels, car en fait ce sont des surcouches logiciels de contrôles de base du système d'exploitation (qui est lui-même fenêtré et se présente sous forme d'une IHM dénommée bureau électronique).



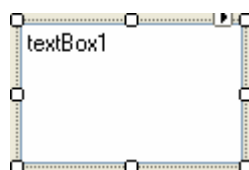
un bouton



Un menu



une boîte de saisie mono-ligne



une boîte de saisie multi-ligne

Les données sont présentées à un instant précis du dialogue à travers des **objets de sortie** de l'information (boîte d'édition monoligne, multiligne, tableaux, graphiques, images, sons etc...).

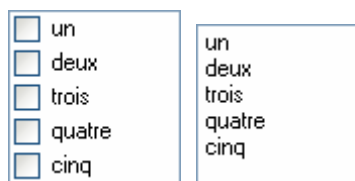
Ci-dessous quelques objets visuels associés à des objets de sortie de l'information :



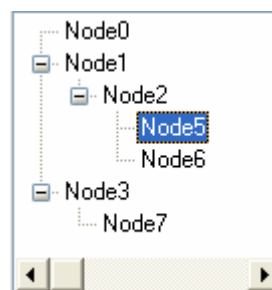
un dataGridView (table)



un pictureBoxe(image jpg, png, bm ,ico,...)



un checkedListBox et un listBox (listes)



un treeView (arbre)

Concept

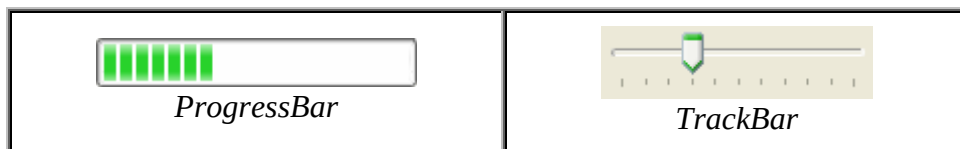
Les temps d'attente

Sur cette question, une approche psychologique est la seule réponse possible, car l'impression d'attente ne dépend que de celui qui attend selon son degré de patience. Toutefois, puisque nous avons parlé de la mémoire à court terme (mémoire rapide), nous pouvons nous baser sur les temps de persistance généralement admis (environ 5 secondes).

Nous considérerons qu'en première approximation, si le délai d'attente est :

- ❑ inférieur à environ une seconde la réponse est quasi-instantanée,
- ❑ compris entre une seconde et cinq secondes il y a attente, toutefois la mémoire rapide de l'utilisateur contient encore la finalité de l'opération en cours.
- ❑ lorsque l'on dépasse la capacité de mémorisation rapide de l'utilisateur alors il faut soutenir l'attention de l'utilisateur en lui envoyant des informations sur le déroulement de l'opération en cours (on peut utiliser pour cela par exemple des barres de défilement, des jauges, des boîtes de dialogue, etc...)

Exemples de quelques classes d'objets visuels de gestion du délai d'attente :



statusStrip (avec deux éléments ici)

Concept

Le pilotage de l'utilisateur

Nous ne cherchons pas à explorer les différentes méthodes utilisables pour piloter un utilisateur dans sa navigation dans une interface. Nous adoptons plutôt la position du concepteur de logiciel qui admet que le futur utilisateur ne se servira de son logiciel que d'une façon épisodique. Il n'est donc pas question de demander à l'utilisateur de connaître en permanence toutes les fonctionnalités du logiciel.

En outre, il ne faut pas non plus submerger l'utilisateur de conseils de guides et d'aides à profusion, car ils risqueraient de le détourner de la finalité du logiciel. Nous préférons adopter une ligne moyenne qui consiste à fournir de petites aides rapides contextuelles (au moment où l'utilisateur en a besoin) et une aide en ligne générale qu'il pourra consulter s'il le souhaite.

Ce qui revient à dire que l'on accepte deux niveaux de navigation dans un logiciel :

- le niveau de surface permettant de réagir aux principales situations,

- le niveau approfondi qui permet l'utilisation plus complète du logiciel.

Il faut admettre que le niveau de surface est celui qui restera le plus employé (l'exemple d'un logiciel de traitement de texte courant du commerce montre qu'au maximum 30% des fonctionnalités du produit sont utilisées par plus de 90% des utilisateurs).

Pour permettre un pilotage plus efficace on peut établir à l'avance un graphe d'actions possibles du futur utilisateur (nous nous servirons du graphe événementiel) et ensuite diriger l'utilisateur dans ce graphe en matérialisant (masquage ou affichage) les actions qui sont réalisables. En application des notions acquises dans les chapitres précédents, nous utiliserons un pilotage dirigé par la syntaxe comme exemple.

Les types d'interaction

Concept

Le tout premier genre d'interaction entre l'utilisateur et un logiciel est apparu sur les premiers systèmes d'exploitation sous la forme d'un langage de commande. L'utilisateur dispose d'une famille de commandes qu'il est censé connaître, le logiciel étant doté d'une interface interne (l'interpréteur de cette famille de commandes). Dès que l'utilisateur tape textuellement une commande (exemple MS-DOS " **dir c: /w** "), le système l'interprète (dans l'exemple : lister en prenant toutes les colonnes d'écran, les bibliothèques et les fichiers du disque C).

Nous adoptons comme mode d'interaction entre un utilisateur et un logiciel, une extension plus moderne de ce genre de dialogue, en y ajoutant, en privilégiant, la notion d'objets visuels permettant d'effectuer des commandes par actions et non plus seulement par syntaxe textuelle pure.

Nous construisons donc une interface tout d'abord **essentiellement** à partir des interactions événementielles, puis lorsque cela est utile ou nécessaire, nous pouvons ajouter un interpréteur de langage (nous pouvons par exemple utiliser des automates d'états finis pour la reconnaissance).

L'enchaînement des opérations

Concept

Nous savons que nous travaillons sur des machines de Von Neumann, donc séquentielles, les opérations internes s'effectuant selon un ordre unique sur lequel l'utilisateur n'a aucune prise. L'utilisateur est censé pouvoir agir d'une manière " aléatoire ". Afin de simuler une certaine liberté d'action de l'utilisateur nous lui ferons parcourir un graphe événementiel prévu par le programmeur. Il y a donc contradiction entre la rigidité séquentielle imposée par la machine et la liberté d'action que l'on souhaite accorder à l'utilisateur. Ce problème est déjà présent dans un système d'exploitation et il relève de la notion de gestion des interruptions.

Nous pouvons trouver un compromis raisonnable dans le fait de découper les tâches internes en tâches séquentielles minimales ininterrompibles et en tâches interrompibles.

Les interruptions consisteront en actions potentielles de l'utilisateur sur la tâche en cours afin de :

- ❑ interrompre le travail en cours,
- ❑ quitter définitivement le logiciel,
- ❑ interroger un objet de sortie,
- ❑ lancer une commande exploratoire ...

Il faut donc qu'existe dans le système de développement du logiciel, un mécanisme qui permette de " *demander la main au système* " sans arrêter ni bloquer le reste de l'interface, ceci pendant le déroulement d'une action répétitive et longue. Lorsque l'interface a la main, l'utilisateur peut alors interrompre, quitter, interroger...

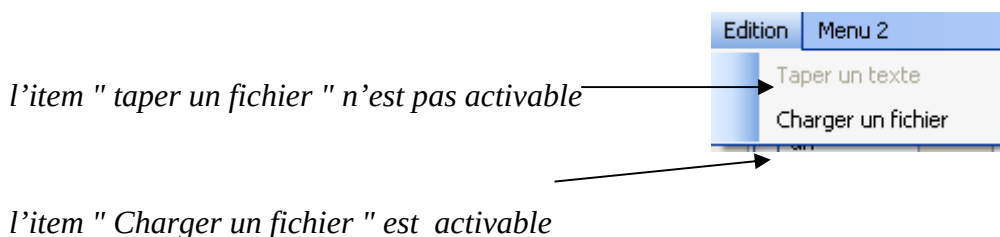
Ce mécanisme est disponible dans les RAD visuels pédagogiques (Delphi, Visual Basic, Visual C#), nous verrons comment l'implanter. Terminons ce tour d'horizon, par le dernier concept de base d'une interface : sa capacité à absorber certains dysfonctionnements.

Concept

La résistance aux erreurs

Il faut en effet employer une méthode de programmation défensive afin de protéger le logiciel contre des erreurs comme par exemple des erreurs de manipulation de la part de l'utilisateur. Nous utilisons plusieurs outils qui concourent à la robustesse de notre logiciel. La protection est donc située à plusieurs niveaux.

1°) Une protection est apportée par le graphe événementiel qui n'autorise que certaines actions (activation-désactivation), matérialisé par un objet tel qu'un menu :



2°) Une protection est apportée par le filtrage des données (on peut utiliser par exemple des logiciels d'automates de reconnaissance de la syntaxe des données).

3°) Un autre niveau de protection est apporté par les composants visuels utilisés qui sont sécurisés dans le RAD à l'origine. Par exemple la méthode LoadFile d'un richTextBox de C# qui permet le chargement d'un fichier dans un composant réagit d'une manière sécuritaire (c'est à dire rien ne se produit) lorsqu'on lui fournit un chemin erroné ou que le fichier n'existe pas.

4°) Un niveau de robustesse est apporté en C# par une utilisation des exceptions (semblable à C++ ou à Delphi) autorisant le détournement du code afin de traiter une situation interne anormale (dépassement de capacité d'un calcul, transtypage de données non conforme etc...). Le programmeur peut donc prévoir les incidents possibles et construire des gestionnaires d'exceptions.

Les événements avec



Plan général:

1. Construction de nouveaux événements

- Design Pattern observer
- Abonné à un événement
- Déclaration d'un événement
- Invocation d'un événement
- Comment s'abonner à un événement
- Restrictions et normalisation
- Événement normalisé avec informations
- Événement normalisé sans information

2. Les événements dans les Windows.Forms

- Contrôles visuels et événements
- Événement Paint : avec information
- Événement Click : sans information
- Code C# généré

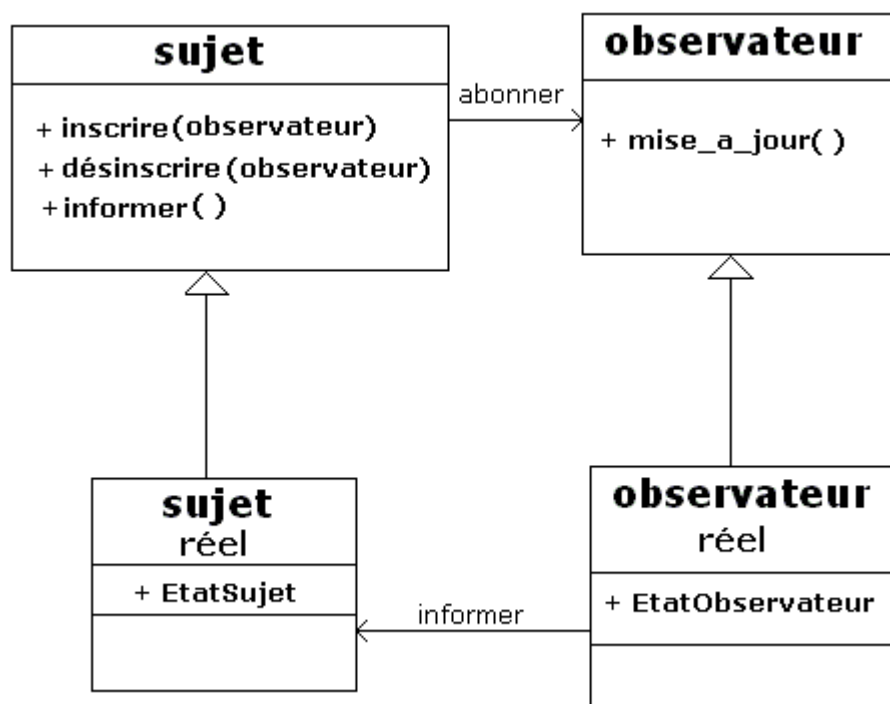
Rappel

Programmation orientée événements

Un programme objet orienté événements est construit avec des objets possédant des propriétés telles que les interventions de l'utilisateur sur les objets du programme et la liaison dynamique du logiciel avec le système déclenchent l'exécution de routines associées. Le traitement en programmation événementielle, consiste à mettre en place un mécanisme d'inteception puis de gestion permettant d'informer un ou plusieurs objets de la survenue d'un événement particulier.

1. Construction de nouveaux événements

Le modèle de conception de l'**observateur** (Design Pattern observer) est utilisé par Java et C# pour gérer un événement. Selon ce modèle, un client s'inscrit sur une liste d'abonnés auprès d'un observateur qui le préviendra lorsqu'un événement aura eu lieu. Les clients délèguent ainsi l'interception d'un événement à une autre entité. Java utilise ce modèle sous forme d'objet écouteur.



Design Pattern observer

Dans l'univers des Design Pattern on utilise essentiellement le modèle observateur dans les cas suivants :

- Quand le changement d'un objet se répercute vers d'autres.
- Quand un objet doit prévenir d'autres objets sans pour autant les connaître.

C# propose des mécanismes supportant les événements, mais avec une implémentation totalement différente de celle de Java. En observant le fonctionnement du langage nous pouvons dire que C# combine efficacement les fonctionnalités de Java et de Delphi.

Abonné à un événement

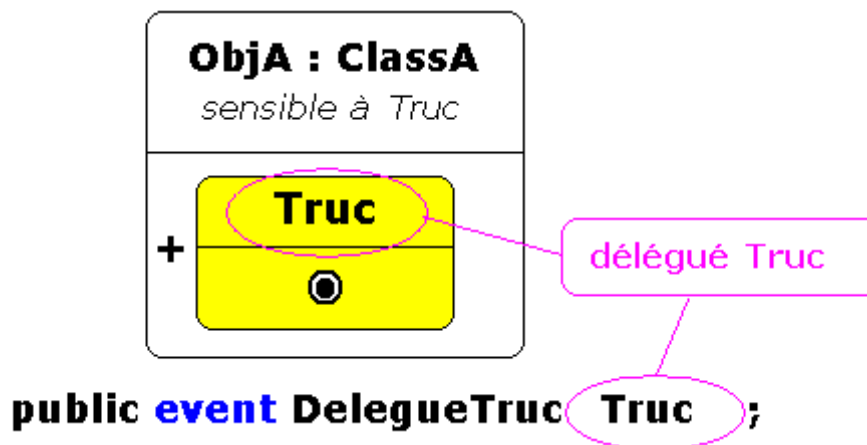
C# utilise les délégués pour fournir un mécanisme explicite permettant de gérer l'abonnement/notification.

En C# la délégation de l'écoute (gestion) d'un événement est confiée à un objet de type délégué : l'abonné est alors une méthode appelée gestionnaire de l'événement (contrairement à Java où l'abonné est une classe) acceptant les mêmes arguments et paramètres de retour que le délégué.

Déclaration d'un événement

type délégation :

```
public Delegate void DelegateTruc( ... ) ;
```



Code C# :

```
using System;
using System.Collections;

namespace ExempleEvent {
    //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string )
    public delegate void DelegateTruc (string s);

    public class ClassA {
        //--> déclaration d'une référence event de type délégué :
        public event DelegateTruc Truc;
    }
}
```

Invocation d'un événement

Une fois qu'une classe a déclaré un événement Truc, elle peut traiter cet événement exactement comme un délégué ordinaire. La démarche est très semblable à celle de Delphi, le champ Truc vaudra **null** si le client ObjA de ClassA n'a pas raccordé un délégué à l'événement Truc. En effet être sensible à plusieurs événements n'oblige pas chaque objet à gérer tous les événements, dans le cas où un objet ne veut pas gérer un événement Truc on n'abonne aucune méthode au délégué Truc qui prend alors la valeur **null**.

Dans l'éventualité où un objet doit gérer un événement auquel il est sensible, il doit invoquer l'événement Truc (qui référence une ou plusieurs méthodes). Invoquer un événement Truc consiste généralement à vérifier d'abord si le champ Truc est null, puis à appeler l'événement (le délégué Truc).

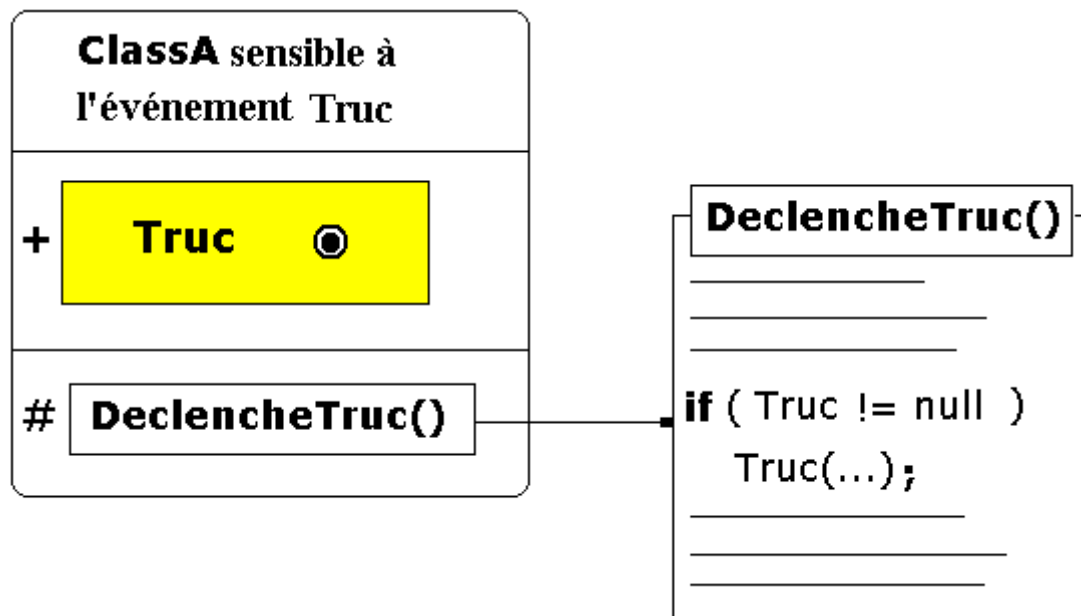
Remarque importante

L'appel d'un événement ne peut être effectué qu'à partir de la classe qui a déclaré cet événement.

Exemple construit pas à pas

Considérons ci-dessous la classe ClassA qui est sensible à un événement que nous nommons Truc (on déclare la référence Truc), dans le corps de la méthode **void DeclencheTruc()** on appelle l'événement Truc.

Nous déclarons cette méthode **void DeclencheTruc()** comme **virtuelle** et **protégée**, de telle manière qu'elle puisse être redéfinie dans la suite de la hiérarchie; ce qui constitue un gage d'évolutivité des futures classes quant à leur comportement relativement à l'événement Truc :



Il nous faut aussi prévoir une méthode **publique** qui permettra d'invoquer l'événement depuis une autre classe, nous la nommons **LancerTruc**.

Code C# , construisons progressivement notre exemple, voici les premières lignes du code :

```

using System;
using System.Collections;

namespace ExempleEvent {
    //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string)
    public delegate void DelegateTruc (string s);

    public class ClassA {
        //--> déclaration d'une référence event de type délégué :
        public event DelegateTruc Truc;
        protected virtual void DeclencheTruc ( ) {
            ....
            if ( Truc != null ) Truc("événement déclenché");
            ....
        }
        public void LancerTruc ( ) {
            ....
            DeclencheTruc ( );
            ....
        }
    }
}

```

Comment s'abonner (se raccorder, s'inscrire, ...) à un événement

Un événement ressemble à un champ public de la classe qui l'a déclaré. Toutefois l'utilisation de ce champ est très restrictive, c'est pour cela qu'il est déclaré avec le spécificateur **event**. Seulement deux opérations sont possibles sur un champ d'événement qui rapellons-le est un délégué :

- Ajouter une nouvelle méthode (à la liste des méthodes abonnées à l'événement).
- Supprimer une méthode de la liste (désabonner une méthode de l'événement).

Enrichissons le code C# précédent avec la classe ClasseUse :

```

using System;
using System.Collections;

namespace ExempleEvent {
    //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string)
    public delegate void DelegateTruc (string s);

    public class ClassA {
        //--> déclaration d'une référence event de type délégué :
        public event DelegateTruc Truc;

        protected virtual void DeclencheTruc ( ) {
            ....
            if ( Truc != null ) Truc("événement déclenché");
            ....
        }
        public void LancerTruc ( ) {
            ....
            DeclencheTruc ( );
            ....
        }
    }

    public class ClasseUse {

```

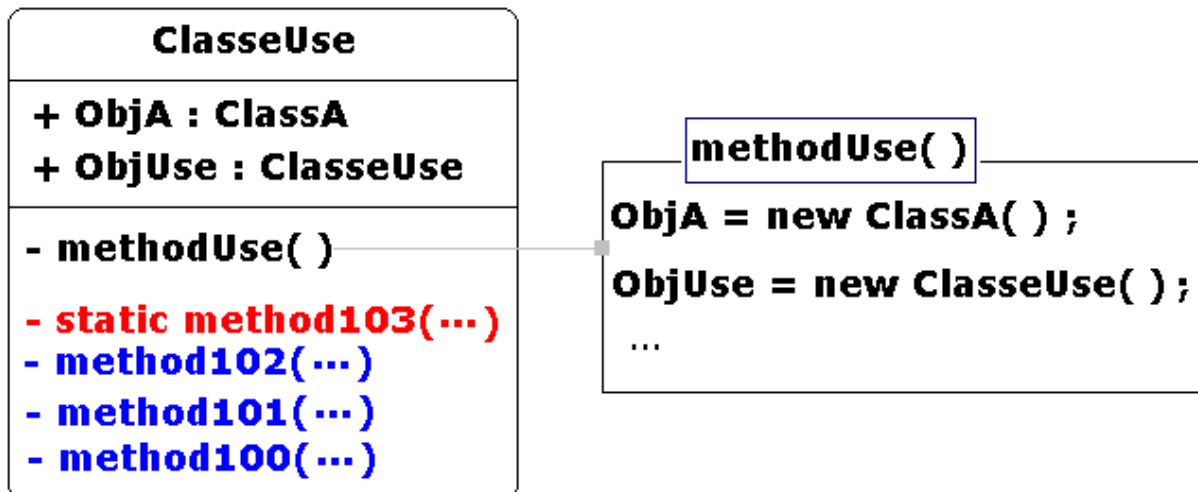
```

static private void methodUse( ) {
    ClassA  ObjA = new ClassA( );
    ClasseUse ObjUse = new ClasseUse ( );
    //....
}
static public void Main(string[] x) {
    methodUse( );
    //....
}
}

```

Il faut maintenant définir des gestionnaires de l'événement Truc (des méthodes ayant la même signature que le type délégation " **public delegate void DelegateTruc (string s);** ". Ensuite nous ajouterons ces méthodes au délégué Truc (nous les abonnerons à l'événement Truc), ces méthodes peuvent être de classe ou d'instance.

Supposons que nous ayons une méthode de classe et trois méthodes d'instances qui vont s'inscrire sur la liste des abonnés à Truc, ce sont quatre gestionnaires de l'événement Truc :



Ajoutons au code C# de la classe ClassA les quatre gestionnaires :

```

using System;
using System.Collections;

namespace ExempleEvent {
    //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string)
    public delegate void DelegateTruc (string s);

    public class ClassA {
        //--> déclaration d'une référence event de type délégué :
        public event DelegateTruc Truc;

        protected virtual void DéclencheTruc( ) {
            ....
            if ( Truc != null ) Truc("événement déclenché")
            ....
        }

        public void LancerTruc( ) {
            ....
            DéclencheTruc( );
        }
    }
}

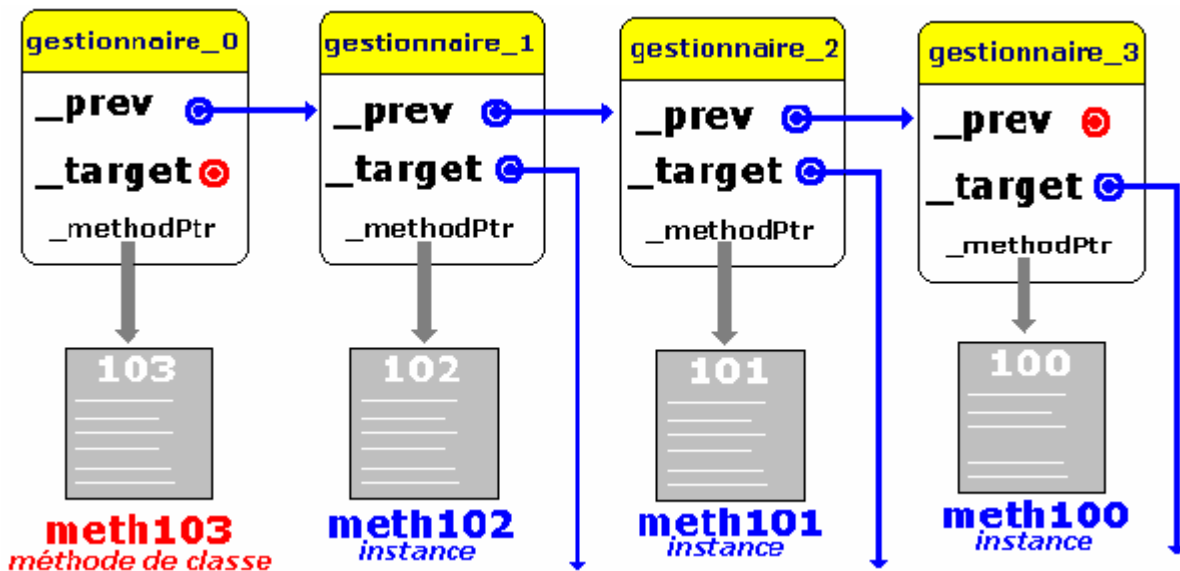
```

```

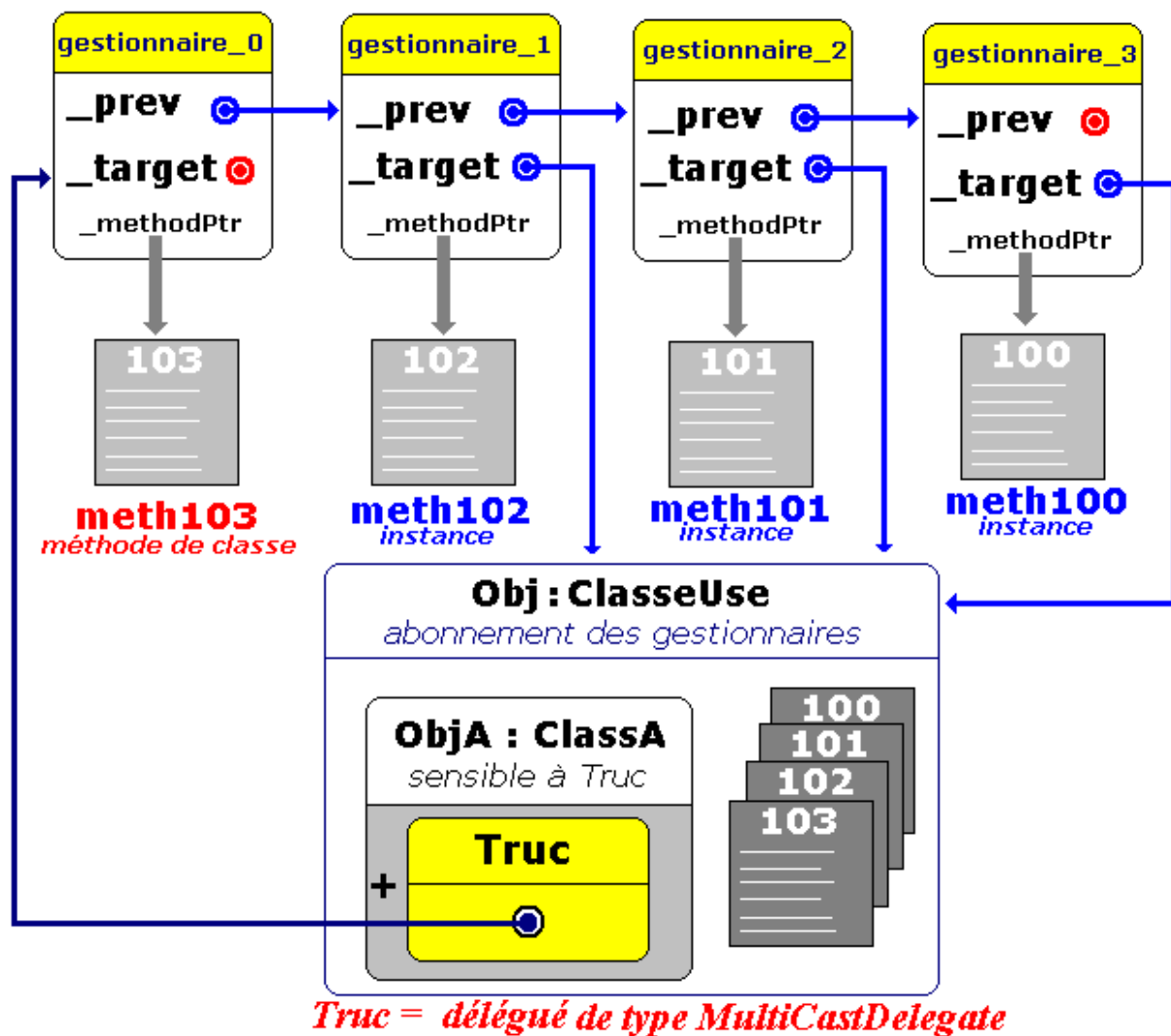
    ....
}
}
public class ClasseUse {
    public void method100(string s); {
        //...gestionnaire d'événement Truc: méthode d'instance.
    }
    public void method101(string s); {
        //...gestionnaire d'événement Truc: méthode d'instance.
    }
    public void method102(string s); {
        //...gestionnaire d'événement Truc: méthode d'instance.
    }
    static public void method103(string s); {
        //...gestionnaire d'événement Truc: méthode de classe.
    }
    static private void methodUse( ) {
        ClassA ObjA = new ClassA( );
        ClasseUse ObjUse = new ClasseUse ( );
        //... il reste à abonner les gestionnaires de l'événement Truc
    }
    static public void Main(string[ ] x) {
        methodUse( );
    }
}
}

```

Lorsque nous ajoutons en C# les nouvelles méthodes method100, ... , method103 au délégué Truc, par surcharge de l'opérateur +, nous dirons que les gestionnaires method100,...,method103, s'abonnent à l'événement Truc.



Prévenir (informer) un abonné correspond ici à l'action d'appeler l'abonné (appeler la méthode) :



Terminons le code C# associé à la figure précédente :

Nous complétons le corps de la méthode " **static private void** methodUse() " par l'abonnement au délégué des quatre gestionnaires.

Pour invoquer l'événement Truc, il faut pouvoir appeler enfin à titre d'exemple une invocation de l'événement :

```
using System;
using System.Collections;

namespace ExempleEvent {
    //--> déclaration du type délégation :(par exemple procédure avec 1 paramètre string)
    public delegate void DelegateTruc (string s);

    public class ClassA {
        //--> déclaration d'une référence event de type délégué :
        public event DelegateTruc Truc;

        protected virtual void DeclencheTruc() {
            //....
        }
    }
}
```

```

        if ( Truc != null ) Truc("événement Truc déclenché");
        //....
    }
    public void LancerTruc() {
        //....
        DeclencheTruc();
        //....
    }
}
public class ClasseUse {
    public void method100(string s) {
        //...gestionnaire d'événement Truc: méthode d'instance.
        System.Console.WriteLine("information utilisateur : "+s);
    }
    public void method101(string s) {
        //...gestionnaire d'événement Truc: méthode d'instance.
        System.Console.WriteLine("information utilisateur : "+s);
    }
    public void method102(string s) {
        //...gestionnaire d'événement Truc: méthode d'instance.
        System.Console.WriteLine("information utilisateur : "+s);
    }
    static public void method103(string s) {
        //...gestionnaire d'événement Truc: méthode de classe.
        System.Console.WriteLine("information utilisateur : "+s);
    }
    static private void methodUse() {
        ClassA ObjA = new ClassA();
        ClasseUse ObjUse = new ClasseUse();
        //-- abonnement des gestionnaires:
        ObjA.Truc += new DelegateTruc ( ObjUse.method100 );
        ObjA.Truc += new DelegateTruc ( ObjUse.method101 );
        ObjA.Truc += new DelegateTruc ( ObjUse.method102 );
        ObjA.Truc += new DelegateTruc ( method103 );
        //-- invocation de l'événement:
        ObjA.LancerTruc(); //...l'appel à cette méthode permet d'invoquer l'événement Truc
    }
    static public void Main(string[] x) {
        methodUse();
    }
}
}

```

Restrictions et normalisation .NET Framework

Bien que le langage C# autorise les événements à utiliser n'importe quel type délégué, le .NET Framework applique à ce jour à des fins de normalisation, certaines indications plus restrictives quant aux types délégués à utiliser pour les événements.

Les indications .NET Framework spécifient que le type délégué utilisé pour un événement doit disposer de deux paramètres et d'un retour définis comme suit :

- un paramètre de type **Object** qui désigne la source de l'événement,
- un autre paramètre soit de classe **EventArgs**, soit d'une classe qui **dérive** de **EventArgs**, il encapsule toutes les informations personnelles relatives à l'événement,
- enfin le type du retour du délégué doit être **void**.

Événement normalisé sans information :

Si vous n'utilisez pas d'informations personnelles pour l'événement, la signature du délégué sera :

```
public delegate void DelegateTruc ( Object sender , EventArgs e ) ;
```

Événement normalisé avec informations :

Si vous utilisez des informations personnelles pour l'événement, vous définirez une classe **MonEventArgs** qui hérite de la classe **EventArgs** et qui contiendra ces informations personnelles, dans cette éventualité la signature du délégué sera :

```
public delegate void DelegateTruc ( Object sender , MonEventArgs e ) ;
```

Il est conseillé d'utiliser la représentation normalisée d'un événement comme les deux exemples ci-dessous le montre, afin d'augmenter la lisibilité et le portage source des programmes événementiels.

Mise en place d'un événement normalisé avec informations

Pour mettre en place un événement Truc normalisé contenant des informations personnalisées vous devrez utiliser les éléments suivants :

- 1°) une classe d'informations personnalisées sur l'événement
- 2°) une déclaration du type délégation normalisée (nom terminé par EventHandler)
- 3°) une déclaration d'une référence Truc du type délégation normalisée spécifiée **event**
- 4.1°) une méthode protégée qui déclenche l'événement Truc (nom commençant par **On**: OnTruc)
- 4.2°) une méthode publique qui lance l'événement par appel de la méthode OnTruc
- 5°) un ou plusieurs gestionnaires de l'événement Truc
- 6°) abonner ces gestionnaires au délégué Truc
- 7°) consommer l'événement Truc

Exemple de code C# directement exécutable, associé à cette démarche :

```
using System;  
using System.Collections;
```

```
namespace ExempleEvent {
```

//--> 1°) classe d'informations personnalisées sur l'événement

```
public class TrucEventArgs : EventArgs {  
    public string info ;  
    public TrucEventArgs (string s) {  
        info = s ;  
    }  
}
```

```
}
```

//--> 2°) déclaration du type délégation normalisé

```
public delegate void DelegateTrucEventHandler ( object sender, TrucEventArgs e );
```

```
public class ClassA {
```

//--> 3°) déclaration d'une référence event de type délégué :

```
public event DelegateTrucEventHandler Truc;
```

//--> 4.1°) méthode protégée qui déclenche l'événement :

```
protected virtual void OnTruc( object sender, TrucEventArgs e ) {  
    //....  
    if ( Truc != null ) Truc( sender , e );  
    //....  
}
```

//--> 4.2°) méthode publique qui lance l'événement :

```
public void LancerTruc() {  
    //....  
    TrucEventArgs evt = new TrucEventArgs ("événement déclenché" );  
    OnTruc ( this , evt );  
    //....  
}
```

```
}
```

```
public class ClasseUse {
```

//--> 5°) les gestionnaires d'événement Truc

```
public void method100( object sender, TrucEventArgs e ){  
    //...gestionnaire d'événement Truc: méthode d'instance.  
    System.Console.WriteLine("information utilisateur 100 : "+e.info);  
}
```

```
public void method101( object sender, TrucEventArgs e ) {  
    //...gestionnaire d'événement Truc: méthode d'instance.  
    System.Console.WriteLine("information utilisateur 101 : "+e.info);  
}
```

```
public void method102( object sender, TrucEventArgs e ) {  
    //...gestionnaire d'événement Truc: méthode d'instance.  
    System.Console.WriteLine("information utilisateur 102 : "+e.info);  
}
```

```
static public void method103( object sender, TrucEventArgs e ) {  
    //...gestionnaire d'événement Truc: méthode de classe.  
    System.Console.WriteLine("information utilisateur 103 : "+e.info);  
}
```

```
static private void methodUse() {  
    ClassA ObjA = new ClassA( );  
    ClasseUse ObjUse = new ClasseUse ( );
```

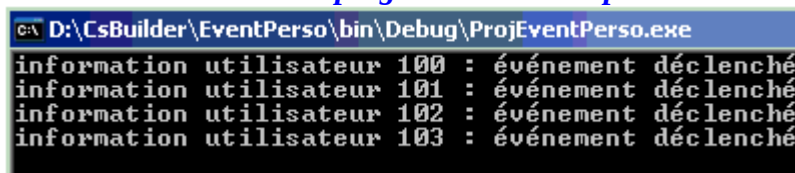
//--> 6°) abonnement des gestionnaires:

```
ObjA.Truc += new DelegateTrucEventHandler ( ObjUse.method100 );  
ObjA.Truc += new DelegateTrucEventHandler ( ObjUse.method101 );  
ObjA.Truc += new DelegateTrucEventHandler ( ObjUse.method102 );  
ObjA.Truc += new DelegateTrucEventHandler ( method103 );
```

//--> 7°) consommation de l'événement:

```
ObjA.LancerTruc( ) ; //...l'appel à cette méthode permet d'invoquer l'événement Truc  
}  
static public void Main(string[] x) {  
    methodUse( ) ;  
}  
}
```

Résultats d'exécution du programme console précédent :



```
C:\ D:\CsBuilder\EventPerso\bin\Debug\ProjEventPerso.exe  
information utilisateur 100 : événement déclenché  
information utilisateur 101 : événement déclenché  
information utilisateur 102 : événement déclenché  
information utilisateur 103 : événement déclenché
```

Mise en place d'un événement normalisé sans information

En fait, pour les événements qui n'utilisent pas d'informations supplémentaires personnalisées, le .NET Framework a déjà défini un type délégué approprié : **System.EventHandler** (équivalent au TNotifyEvent de Delphi):

```
public delegate void EventHandler ( object sender, EventArgs e );
```

Pour mettre en place un événement Truc normalisé sans information spéciale, vous devrez utiliser les éléments suivants :

- 1°) la classe *System.EventArgs*
- 2°) le type délégué normalisé *System.EventHandler*
- 3°) une déclaration d'une référence Truc du type délégué normalisé spécifiée **event**
- 4.1°) une méthode protégée qui déclenche l'événement Truc (nom commençant par **On**: *OnTruc*)
- 4.2°) une méthode publique qui lance l'événement par appel de la méthode *OnTruc*
- 5°) un ou plusieurs gestionnaires de l'événement Truc
- 6°) abonner ces gestionnaires au délégué Truc
- 7°) consommer l'événement Truc

Remarque, en utilisant la déclaration **public delegate void EventHandler (object sender, EventArgs e)** contenue dans **System.EventHandler**, l'événement n'ayant aucune donnée personnalisée, le deuxième paramètre n'étant pas utilisé, il est possible de fournir le champ **static Empty** de la classe **EventArgs** .

EventArgs, membres

Constructeurs publics

EventArgs, constructeur

Initialise une nouvelle instance de la classe **EventArgs**.

Champs publics

Empty

Représente un événement sans données d'événement.

Exemple de code C# directement exécutable, associé à un événement sans information personnalisée :

```
using System;
using System.Collections;
```

```
namespace ExempleEvent {
```

//--> 1°) classe d'informations personnalisées sur l'événement

System.EventArgs est déjà déclarée dans using System;

//--> 2°) déclaration du type délégation normalisé

System.EventHandler est déjà déclarée dans using System;

```
public class ClassA {
```

//--> 3°) déclaration d'une référence event de type délégué :

```
public event EventHandler Truc;
```

//--> 4.1°) méthode protégée qui déclenche l'événement :

```
protected virtual void OnTruc( object sender, EventArgs e ) {
    //....
    if ( Truc != null ) Truc( sender , e );
    //....
}
```

//--> 4.2°) méthode publique qui lance l'événement :

```
public void LancerTruc( ) {
    //....
    OnTruc( this , EventArgs.Empty );
    //....
}
```

```
}
```

```
public class ClasseUse {
```

//--> 5°) les gestionnaires d'événement Truc

```
public void method100( object sender, EventArgs e ){
```

```

        //...gestionnaire d'événement Truc: méthode d'instance.
        System.Console.WriteLine("information utilisateur 100 : événement déclenché");
    }

    public void method101( object sender, EventArgs e ) {
        //...gestionnaire d'événement Truc: méthode d'instance.
        System.Console.WriteLine("information utilisateur 101 : événement déclenché");
    }

    public void method102( object sender, EventArgs e ) {
        //...gestionnaire d'événement Truc: méthode d'instance.
        System.Console.WriteLine("information utilisateur 102 : événement déclenché");
    }

    static public void method103( object sender, EventArgs e ) {
        //...gestionnaire d'événement Truc: méthode de classe.
        System.Console.WriteLine("information utilisateur 103 : événement déclenché");
    }
}

```

```

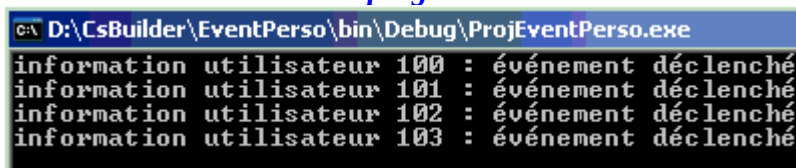
static private void methodUse( ) {
    ClassA ObjA = new ClassA( );
    ClasseUse ObjUse = new ClasseUse ( );
    //--> 6°) abonnement des gestionnaires:
    ObjA.Truc += new DelegateTruc ( ObjUse.method100 );
    ObjA.Truc += new DelegateTruc ( ObjUse.method101 );
    ObjA.Truc += new DelegateTruc ( ObjUse.method102 );
    ObjA.Truc += new DelegateTruc ( method103 );

    //--> 7°) consommation de l'événement:
    ObjA.LancerTruc( ); //...l'appel à cette méthode permet d'invoquer l'événement Truc
}

static public void Main(string[] x) {
    methodUse( );
}
}

```

Résultats d'exécution de ce programme :



```

D:\CsBuilder\EventPerso\bin\Debug\ProjEventPerso.exe
information utilisateur 100 : événement déclenché
information utilisateur 101 : événement déclenché
information utilisateur 102 : événement déclenché
information utilisateur 103 : événement déclenché

```

2. Les événements dans les Windows.Forms

Le code et les copies d'écran sont effectuées avec C#Builder 1.0 de Borland version personnelle gratuite.

Contrôles visuels et événements

L'architecture de fenêtres de .Net FrameWork se trouve essentiellement dans l'espace de noms

System.Windows.Forms qui contient des classes permettant de créer des applications contenant des IHM (interface humain machine) et en particulier d'utiliser les fonctionnalités afférentes aux IHM de Windows.

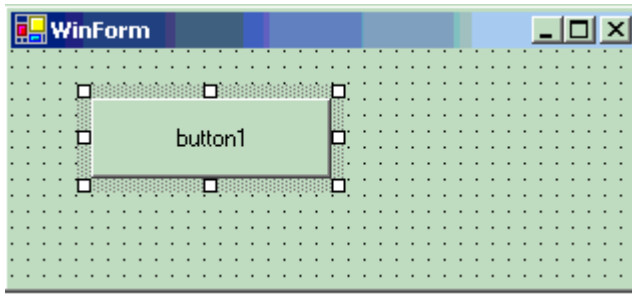
Plus spécifiquement, la classe **System.Windows.Forms.Control** est la classe mère de tous les composants visuels. Par exemple, les classes **Form**, **Button**, **TextBox**, etc... sont des descendants de la classe **Control** qui met à disposition du développeur C# 58 événements auxquels un contrôle est sensible.

Ces 58 événements sont tous normalisés, certains sont des événements sans information spécifique, d'autres possèdent des informations spécifiques, ci-dessous un extrait de la liste des événements de la classe **Control**, plus particulièrement les événements traitant des actions de souris :

Control : Événements publics	
 BackColorChanged	Se produit lorsque la valeur de la propriété BackColor change.
 BackgroundImageChanged	Se produit lorsque la valeur de la propriété BackgroundImage change.
.....	
 Click	Se produit suite à un clic sur le contrôle.
.....	
 MouseDown	Se produit lorsque le pointeur de la souris se trouve sur le contrôle et qu'un bouton de la souris est enfoncé.
 MouseEnter	Se produit lorsque le pointeur de la souris se place dans le contrôle.
 MouseHover	Se produit lorsque la souris pointe sur le contrôle.
 MouseLeave	Se produit lorsque le pointeur de la souris s'écarte du contrôle.
 MouseMove	Se produit lorsque le pointeur de la souris est placé sur le contrôle.
 MouseUp	Se produit lorsque le pointeur de la souris se trouve sur le contrôle et qu'un bouton de la souris est relâché.
 MouseWheel	Se produit lorsque la roulette de la souris bouge alors que le contrôle a le focus.
 Move	Se produit lorsque le contrôle est déplacé.
 Paint	Se produit lorsque le contrôle est redessiné.
.....	

Nous avons mis en évidence deux événements **Click** et **Paint** dont l'un est sans information (**Click**), l'autre est avec information (**Paint**). Afin de voir comment nous en servir nous traitons un exemple :

Soit une fiche (classe **Form1** héritant de la classe **Form**) sur laquelle est déposé un bouton poussoir (classe **Button**) de nom **button1** :



Montrons comment nous programmons la réaction du bouton à un click de souris et à son redessinement. Nous devons faire réagir **button1** qui est sensible à au moins 58 événements, aux deux événements Click et Paint. Rappelons la liste méthodologique ayant trait au cycle événementiel, on doit utiliser :

- 1°) une classe d'informations personnalisées sur l'événement
- 2°) une déclaration du type délégation normalisée (nom terminé par *EventHandler*)
- 3°) une déclaration d'une référence *Truc* du type délégation normalisée spécifiée **event**
- 4.1°) une méthode protégée qui déclenche l'événement *Truc* (nom commençant par **On**: *OnTruc*)
- 4.2°) une méthode publique qui lance l'événement par appel de la méthode *OnTruc*
- 5°) un ou plusieurs gestionnaires de l'événement *Truc*
- 6°) abonner ces gestionnaires au délégué *Truc*
- 7°) consommer l'événement *Truc*

Les étapes 1° à 4° ont été conçues et développées par les équipes de .Net et ne sont plus à notre charge.

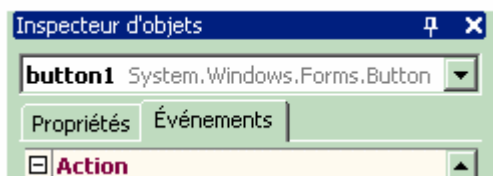
Il nous reste les étapes suivantes :

- 5°) à construire un gestionnaire de réaction de **button1** à l'événement Click et un gestionnaire de réaction de **button1**
- 6°) à abonner chaque gestionnaire au délégué correspondant (Click ou Paint)

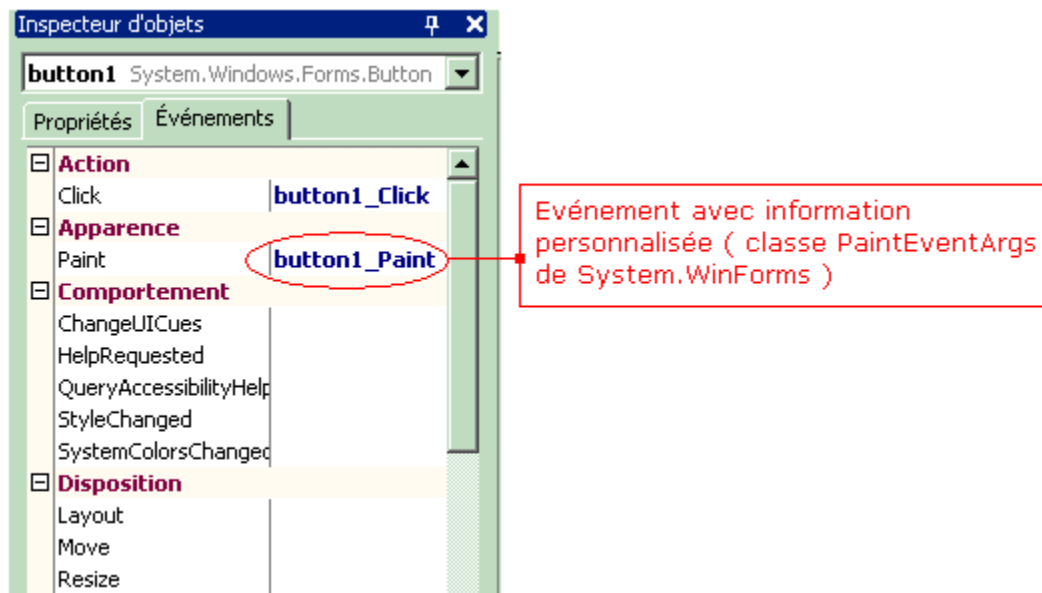
L'étape 7° est assurée par le système d'exploitation qui se charge d'envoyer des messages et de lancer les événements.

Événement Paint : normalisé avec informations

Voici dans l'inspecteur d'objets de C#Builder l'onglet Événements qui permet de visualiser le délégué à utiliser ainsi que le gestionnaire à abonner à ce délégué.



Dans le cas de l'événement Paint, le délégué est du type **PaintEventArgs** situé dans **System.Windows.Forms** :



signature du délégué Paint dans System.Windows.Forms :

```
public delegate void PaintEventHandler ( object sender, PaintEventArgs e );
```

La classe **PaintEventArgs** :

PaintEventArgs

Constructeur public

[PaintEventArgs, constructeur](#)
Initialise une nouvelle instance de la classe **PaintEventArgs** avec les graphiques et le rectangle de découpage spécifiés.

Propriétés publiques

[ClipRectangle](#)
Obtient le rectangle dans lequel peindre.

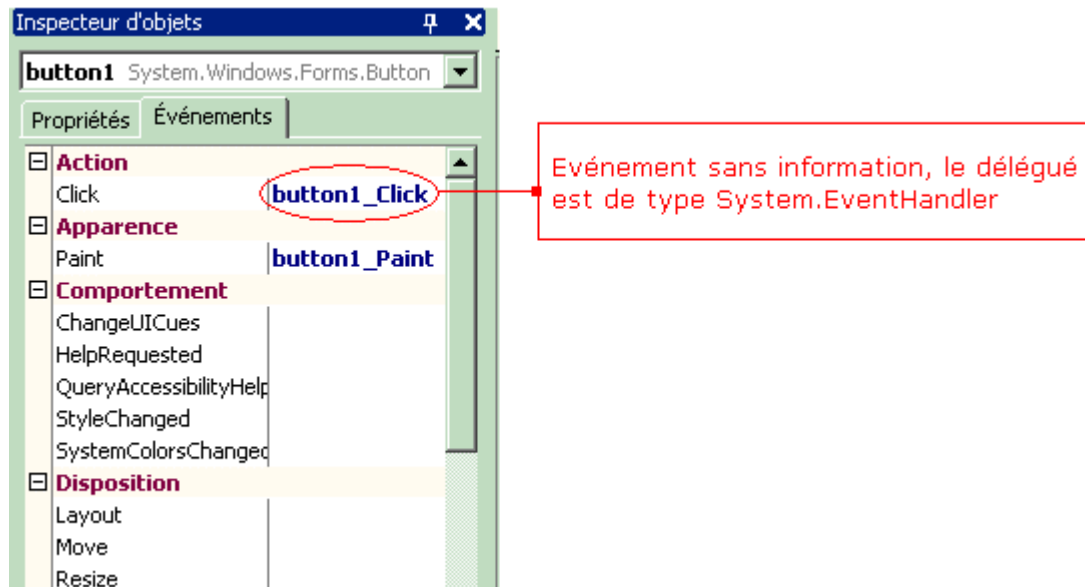
[Graphics](#)
Obtient le graphique utilisé pour peindre.

Méthodes publiques

[Dispose](#)
Surchargé. Libère les ressources utilisées par **PaintEventArgs**.

Événement Click normalisé sans information

Dans le cas de l'événement Click, le délégué est de type **Event Handler** situé dans System :



signature du délégué Click dans System :

```
public delegate void EventHandler ( object sender, EventArgs e );
```

En fait l'inspecteur d'objet de C#Builder permet de réaliser en mode visuel la machinerie des étapes qui sont à notre charge :

- le délégué de l'événement, [**public event** EventHandler Click;]
- le squelette du gestionnaire de l'événement, [**private void** button1_Click(object sender, EventArgs e){ }]
- l'abonnement de ce gestionnaire au délégué. [**this.button1.Click += new** System.EventHandler (**this.button1_Click**);]



Code C# généré

Voici le code généré par C#Builder utilisé en conception visuelle pour faire réagir **button1** aux deux événements Click et Paint :

```

public class WinForm : System.Windows.Forms.Form {
    private System.ComponentModel.Container components = null ;
    private System.Windows.Forms.Button button1;

    public WinForm() {
        InitializeComponent();
    }

    protected override void Dispose (bool disposing) {
        if (disposing) {
            if (components != null) {
                components.Dispose();
            }
        }
        base.Dispose(disposing);
    }
    private void InitializeComponent() {
        this.button1 = new System.Windows.Forms.Button();
        .....

        this.button1.Click += new System.EventHandler( this.button1_Click );

        this.button1.Paint += new System.Windows.Forms.PaintEventHandler( this.button1_Paint );

        .....
    }
    static void Main() {
        Application.Run( new WinForm() );
    }

    private void button1_Click(object sender, System.EventArgs e) {
        //..... gestionnaire de l'événement OnClick
    }

    private void button1_Paint(object sender, System.Windows.Forms.PaintEventArgs e) {
        //..... gestionnaire de l'événement OnPaint
    }
}

```

La machinerie événementielle est automatiquement générée par l'environnement RAD, ce qui épargne de nombreuses lignes de code au développeur et le laisse libre de penser au code spécifique de réaction à l'événement.

Propriétés et indexeurs en



Plan général:

1. Les propriétés

- 1.1 Définition et déclaration de propriété
- 1.2 Accesseurs de propriété
- 1.3 Détail et exemple de fonctionnement d'une propriété
 - Exemple du fonctionnement
 - Explication des actions
- 1.4 Les propriétés sont de classes ou d'instances
- 1.5 Les propriétés peuvent être masquées comme les méthodes
- 1.6 Les propriétés peuvent être virtuelles et redéfinies comme les méthodes
- 1.7 Les propriétés peuvent être abstraites comme les méthodes
- 1.8 Les propriétés peuvent être déclarées dans une interface
- 1.9 Exemple complet exécutable
 - 1.9.1 Détail du fonctionnement en écriture
 - 1.9.2 Détail du fonctionnement en lecture

2. Les indexeurs

- 2.1 Définitions et comparaisons avec les propriétés
 - 2.1.1 Déclaration
 - 2.1.2 Utilisation
 - 2.1.3 Paramètres
 - 2.1.4 Liaison dynamique abstraction et interface
- 2.2 Code C# complet compilable

1. Les propriétés

Les propriétés du langage C# sont très proches de celle du langage Delphi, mais elles sont plus complètes et restent cohérentes avec la notion de membre en C#.

1.1 Définition et déclaration de propriété

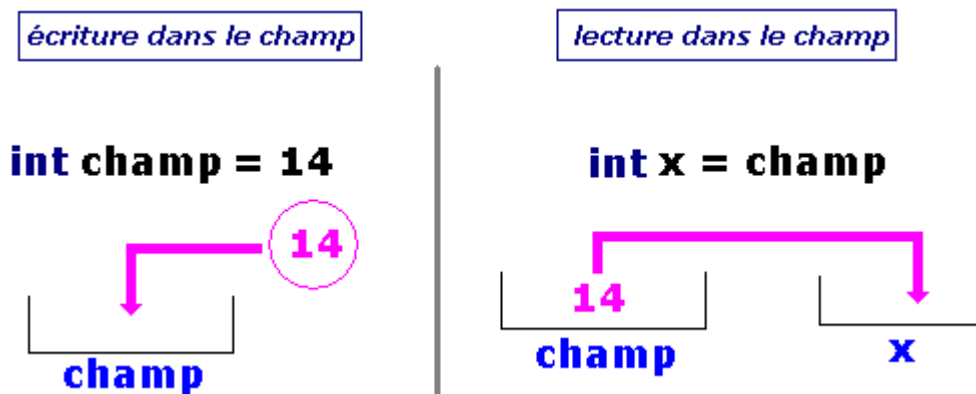
Définition d'une propriété

Une propriété définie dans une classe permet d'accéder à certaines informations contenues dans les objets instanciés à partir de cette classe. Une propriété a la même syntaxe de définition et d'utilisation que celle d'un champ d'objet (elle possède un type de déclaration), mais en fait elle invoque une ou deux méthodes internes pour fonctionner. Les méthodes internes sont déclarées à l'intérieur d'un bloc de définition de la propriété.

Déclaration d'une propriété `prop1` de type `int` :

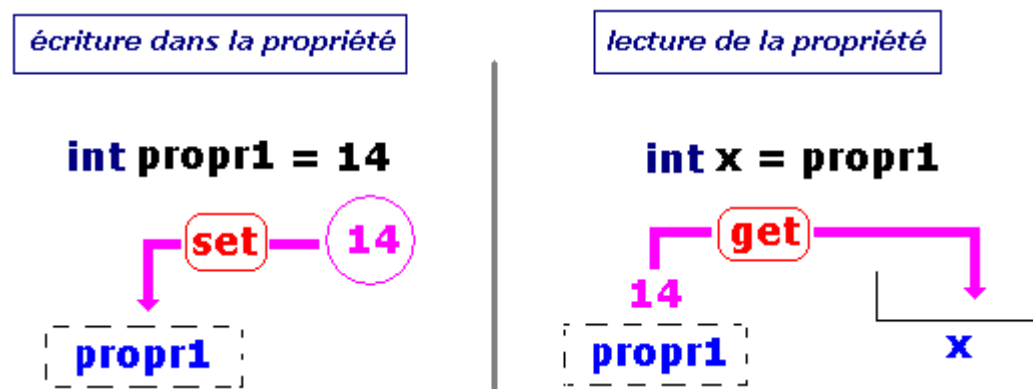
```
public int prop1 {  
    //..... bloc de définition  
}
```

Un champ n'est qu'un emplacement de stockage dont le contenu peut être consulté (**lecture du contenu du champ**) et modifié (**écriture dans le champ**), tandis qu'une propriété associe des **actions spécifiques à la lecture ou à l'écriture** ainsi que la modification des données que la propriété représente.



1.2 Accesseurs de propriété

En C#, une propriété fait systématiquement appel à une ou à deux méthodes internes dont les noms sont les mêmes pour toutes les propriétés afin de fonctionner soit en **lecture**, soit en **écriture**. On appelle ces méthodes internes des **accesseurs**; leur noms sont **get** et **set**, ci-dessous un exemple de lecture et d'écriture d'une propriété au moyen d'affectations :



Accesseur de lecture de la propriété :

Syntaxe : `get { return ..... ; }`

cet accesseur indique que la propriété est en lecture et doit renvoyer un résultat dont le type doit être le même que celui de la propriété. La propriété `propr1` ci-dessous est déclarée en lecture seule et renvoie le contenu d'un champ de même type qu'elle :

```
private int champ;  
public int propr1{  
    get { return champ ; }  
}
```

Accesseur d'écriture dans la propriété :

Syntaxe : `set { .... }`

cet accesseur indique que la propriété est en écriture et sert à initialiser ou à modifier la propriété. La propriété `propr1` ci-dessous est déclarée en écriture seule et stocke une donnée de même type qu'elle dans la variable `champ` :

```
private int champ;  
public int propr1{  
    set { champ = value ; }  
}
```

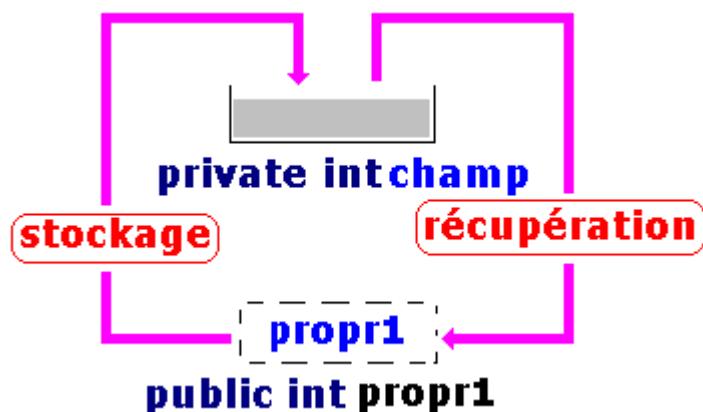
Le mot clef **value** est une sorte de paramètre implicite interne à l'accesseur **set**, il contient la valeur effective qui est transmise à la propriété lors de l'accès en écriture.

D'une manière générale lorsqu'une propriété fonctionne à travers un attribut (du même type que la propriété), l'attribut contient la donnée brute à laquelle la propriété permet d'accéder.

Ci-dessous une déclaration d'une propriété en lecture et écriture avec attribut de stockage :

```
private int champ;  
  
public int propr1{  
    get { return champ ; }  
    set { champ = value ; }  
}
```

Le mécanisme de fonctionnement est figuré ci-après :



Dans l'exemple précédent, la propriété accède directement sans modification à la donnée brute stockée dans le champ, mais il est tout à fait possible à une propriété d'accéder à cette donnée en en modifiant sa valeur avant stockage ou après récupération de sa valeur.

1.3 Détail et exemple de fonctionnement d'une propriété

L'exemple ci-dessous reprend la propriété `propr1` en lecture et écriture du paragraphe précédent et montre comment elle peut modifier la valeur brute de la donnée stockée dans l'attribut " `int champ` " :

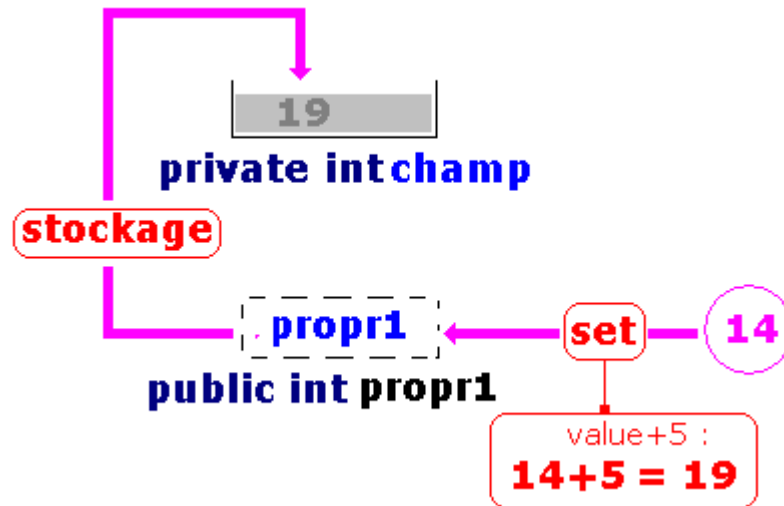
```
private int champ;  
  
public int propr1{  
    get { return champ*10;}  
    set { champ = value + 5 ; }  
}
```

Utilisons cette propriété en mode écriture à travers une affectation :

```
propr1 = 14 ;
```

Le mécanisme d'écriture est simulé ci-dessous :

La valeur 14 est passée comme paramètre dans la méthode **set** à la variable implicite **value**, le calcul $value+5$ est effectué et le résultat 19 est stocké dans l'attribut **champ**.

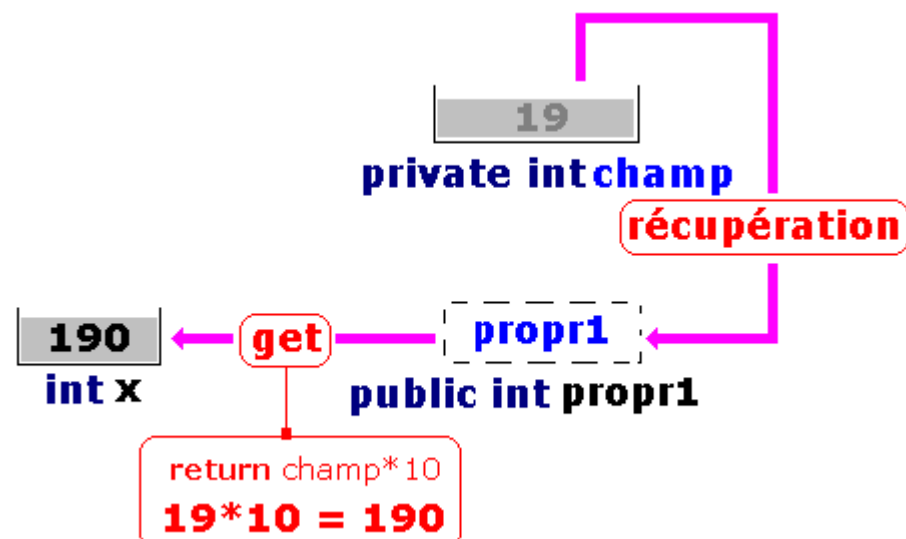


Utilisons maintenant notre propriété en mode lecture à travers une affectation :

```
int x = propr1 ;
```

Le mécanisme de lecture est simulé ci-dessous :

La valeur brute 19 stockée dans l'attribut **champ** est récupérée par la propriété qui l'utilise dans la méthode accesseur **get** en la multipliant par 10, c'est cette valeur modifiée de 190 qui renvoyée par la propriété.



Exemple pratique d'utilisation d'une propriété

Une propriété servant à fournir automatiquement le prix d'un article en y intégrant la TVA au taux de 19.6% et arrondi à l'unité d'euro supérieur :

```
private Double prixTotal ;
private Double tauxTVA = 1.196 ;

public Double prix {
    get {
        return Math.Round(prixTotal);
    }
    set {
        prixTotal = value * tauxTVA ;
    }
}
```

Ci-dessous le programme console C#Builder exécutable :

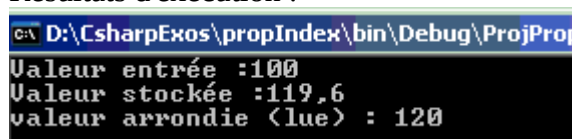
```
using System ;

namespace ProjPropIndex
{
    class Class {
        static private Double prixTotal ;
        static private Double tauxTVA = 1.196 ;

        static public Double prix {
            get {
                return Math.Round ( prixTotal ) ;
            }
            set {
                prixTotal = value * tauxTVA ;
            }
        }

        [STAThread]
        static void Main ( string [] args ) {
            Double val = 100 ;
            System.Console.WriteLine ("Valeur entrée :" + val );
            prix = val ;
            System.Console.WriteLine ("Valeur stockée :" + prixTotal );
            val = prix ;
            System.Console.WriteLine ("valeur arrondie (lue) : " + val );
            System.Console.ReadLine ();
        }
    }
}
```

Résultats d'exécution :



```
C:\D:\CsharpExos\projIndex\bin\Debug\ProjProp
Valeur entrée :100
Valeur stockée :119,6
valeur arrondie (lue) : 120
```

Explications des actions exécutées :

On rentre 100€ dans la variable prix :

```
Double val = 100 ;  
prix = val ;
```

Action effectuée :

On écrit 100 dans la propriété prix et celle-ci stocke $100 \times 1.196 = 119.6$ dans le champ prixTotal.

On exécute l'instruction :

```
val = prix ;
```

Action effectuée :

On lit la propriété qui arrondi le champ prixTotal à l'euro supérieur soit : 120€

1.4 Les propriétés sont de classes ou d'instances

Les propriétés, comme les champs peuvent être des **propriétés de classes** et donc qualifiées par les mots clefs comme **static**, **abstract** etc ... Dans l'exemple précédent nous avons qualifié tous les champs et la propriété prix en **static** afin qu'ils puissent être accessibles à la méthode **Main** qui est elle-même obligatoirement **static**.

Voici le même exemple utilisant une version avec des propriétés et des champs d'instances et non de classe (non static) :

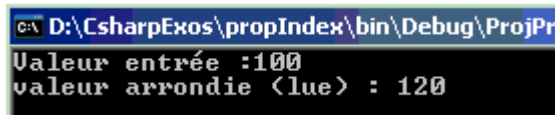
```
using System ;  
  
namespace ProjPropIndex  
{  
    class clA {  
        private Double prixTotal ;  
        private Double tauxTVA = 1.196 ;  
  
        public Double prix {  
            get { return Math.Round ( prixTotal ) ; }  
            set { prixTotal = value * tauxTVA ; }  
        }  
    }  
  
    class Class {  
  
        [STAThread]  
        static void Main ( string [] args ) {  
            clA Obj = new clA () ;  
            Double val = 100 ;
```

```

    System.Console.WriteLine ("Valeur entrée : " + val );
    Obj.prix = val ;
    // le champ prixTotal n'est pas accessible car il est privé
    val = Obj.prix ;
    System.Console.WriteLine ("valeur arrondie (lue) : " + val );
    System.Console.ReadLine ();
}
}
}

```

Résultats d'exécution :



```

C:\D:\CsharpExos\propIndex\bin\Debug\ProjPr
Valeur entrée :100
valeur arrondie <lue> : 120

```

1.5 Les propriétés peuvent être masquées comme les méthodes

Une propriété sans spécificateur particulier de type de liaison est considérée comme une entité à liaison statique par défaut.

Dans l'exemple ci-après nous dérivons une nouvelle classe de la classe clA nommée clB, nous redéclarons dans la classe fille une nouvelle propriété ayant le même nom, à l'instar d'un champ ou d'une méthode C# considère que nous masquons la propriété mère et nous suggère le conseil suivant :

[C# Avertissement] Class..... : Le mot clé new est requis sur '.....', car il masque le membre hérité..... '

Nous mettons donc le mot clef **new** devant la nouvelle déclaration de la propriété dans la classe fille. En reprenant l'exemple précédent supposons que dans la classe fille clB, la TVA soit à 5%, nous redéclarons dans clB une propriété prix qui va masquer celle de la mère :

```

using System ;

namespace ProjPropIndex
{
    class clA {
        private Double prixTotal ;
        private Double tauxTVA = 1.196 ;

        public Double prix { // propriété de la classe mère
            get { return Math.Round ( prixTotal ) ; }
            set { prixTotal = value * tauxTVA ; }
        }
    }
    class clB : clA {
        private Double prixLocal ;
        public new Double prix { // masquage de la propriété de la classe mère
            get { return Math.Round ( prixLocal ) ; }
            set { prixLocal = value * 1.05 ; }
        }
    }
    class Class {

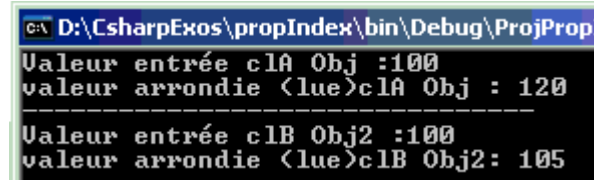
```

```

[STAThread]
static void Main ( string [] args ) {
    clA Obj = new clA ();
    Double val = 100 ;
    System.Console.WriteLine ("Valeur entrée clA Obj : " + val );
    Obj.prix = val ;
    val = Obj.prix ;
    System.Console.WriteLine ("valeur arrondie (lue)clA Obj : " + val );
    System.Console.WriteLine ("-----");
    clB Obj2 = new clB ();
    val = 100 ;
    System.Console.WriteLine ("Valeur entrée clB Obj2 : " + val );
    Obj2.prix = val ;
    val = Obj2.prix ;
    System.Console.WriteLine ("valeur arrondie (lue)clB Obj2: " + val );
    System.Console.ReadLine ();
}
}
}

```

Résultats d'exécution :



```

C:\D:\CsharpExos\propIndex\bin\Debug\ProjProp
Valeur entrée clA Obj :100
valeur arrondie <lue>clA Obj : 120
-----
Valeur entrée clB Obj2 :100
valeur arrondie <lue>clB Obj2: 105

```

1.6 Les propriétés peuvent être virtuelles et redéfinies comme les méthodes

Les propriétés en C# ont l'avantage important d'être utilisables dans le contexte de liaison dynamique d'une manière strictement identique à celle des méthodes en C#, ce qui confère au langage une "orthogonalité" solide relativement à la notion de polymorphisme.

Une propriété peut donc être déclarée virtuelle dans une classe de base et être surchargée dynamiquement dans les classes descendantes de cette classe de base.

Dans l'exemple ci-après semblable au précédent, nous déclarons dans la classe mère **clA** la propriété **prix** comme **virtual**, puis :

- Nous dérivons **clB**, une classe fille de la classe **clA** possédant une propriété **prix** masquant statiquement la propriété virtuelle de la classe **clA**, dans cette classe **clB** la TVA appliquée à la variable **prix** est à 5% (nous mettons donc le mot clef **new** devant la nouvelle déclaration de la propriété **prix** dans la classe fille **clB**). La propriété **prix** est dans cette classe **clB** à liaison statique.
- Nous dérivons une nouvelle classe de la classe **clA** nommée **clB2** dans laquelle nous redéfinissons en **override** la propriété **prix** ayant le même nom, dans cette classe **clB2** la TVA appliquée à la variable **prix** est aussi à 5%. La propriété **prix** est dans cette classe **clB2** à liaison dynamique.

Notre objectif est de comparer les résultats d'exécution obtenus lorsque l'on utilise une référence

d'objet de classe mère instanciée soit en objet de classe **clB** ou **clB2**. C'est le comportement de la propriété **prix** dans chacun de deux cas (statique ou dynamique) qui nous intéresse :

```
using System ;

namespace ProjPropIndex
{
    class clA {
        private Double prixTotal ;
        private Double tauxTVA = 1.196 ;

        public virtual Double prix { // propriété virtuelle de la classe mère
            get { return Math.Round ( prixTotal ) ; }
            set { prixTotal = value * tauxTVA ; }
        }
    }

    class clB : clA {
        private Double prixLocal ;
        public new Double prix { // masquage de la propriété de la classe mère
            get { return Math.Round ( prixLocal ) ; }
            set { prixLocal = value * 1.05 ; }
        }
    }

    class clB2 : clA {
        private Double prixLocal ;
        public override Double prix { // redéfinition de la propriété de la classe mère
            get { return Math.Round ( prixLocal ) ; }
            set { prixLocal = value * 1.05 ; }
        }
    }

    class Class {
        static private Double prixTotal ;
        static private Double tauxTVA = 1.196 ;

        static public Double prix {
            get { return Math.Round ( prixTotal ) ; }
            set { prixTotal = value * tauxTVA ; }
        }
    }

    [STAThread]
    static void Main ( string [] args ) {
        clA Obj = new clA () ;
        Double val = 100 ;
        System.Console.WriteLine ("Valeur entrée Obj=new clA : " + val ) ;
        Obj.prix = val ;
        val = Obj.prix ;
        System.Console.WriteLine ("valeur arrondie (lue)Obj=new clA : " + val ) ;
        System.Console.WriteLine ("-----");
        Obj = new clB () ;
        val = 100 ;
        System.Console.WriteLine ("Valeur entrée Obj=new clB : " + val ) ;
        Obj.prix = val ;
        val = Obj.prix ;
        System.Console.WriteLine ("valeur arrondie (lue)Obj=new clB : " + val ) ;
        System.Console.WriteLine ("-----");
        Obj = new clB2 () ;
        val = 100 ;
        System.Console.WriteLine ("Valeur entrée Obj=new clB2 : " + val ) ;
        Obj.prix = val ;
    }
}
```

```

val = Obj.prix ;
System.Console.WriteLine ("valeur arrondie (lue)Obj=new clB2 : " + val );
System.Console.ReadLine ( );
}
}
}

```

Résultats d'exécution :

```

C:\D:\CsharpExos\propIndex\bin\Debug\ProjPropIndex.exe
Valeur entrée  Obj=new clA :100
valeur arrondie <lue>Obj=new clA : 120
-----
Valeur entrée  Obj=new clB :100
valeur arrondie <lue>Obj=new clB : 120
-----
Valeur entrée  Obj=new clB2 :100
valeur arrondie <lue>Obj=new clB2 : 105

```

Nous voyons bien que le même objet Obj instancié en classe **clB** ou en classe **clB2** ne fournit pas les mêmes résultats pour la propriété prix, ces résultats sont conformes à la notion de polymorphisme en particulier pour l'instanciation en **clB2**.

Rappelons que le **masquage statique** doit être utilisé comme pour les méthodes à bon escient, plus spécifiquement lorsque nous ne souhaitons pas utiliser le polymorphisme, dans le cas contraire c'est la **liaison dynamique** qui doit être utilisée pour **définir et redéfinir des propriétés**.

1.7 Les propriétés peuvent être abstraites comme les méthodes

Les propriétés en C# peuvent être déclarées **abstract**, dans ce cas comme les méthodes elles sont automatiquement virtuelles sans nécessiter l'utilisation du mot clef **virtual**.

Comme une méthode abstraite, une propriété abstraite n'a pas de corps de définition pour le ou les accesseurs qui la composent, ces accesseurs sont implémentés dans une classe fille.

Toute classe déclarant une propriété **abstract** doit elle-même être déclarée **abstract**, l'implémentation de la propriété a lieu dans une classe fille, soit en masquant la propriété de la classe mère (grâce à une déclaration à liaison statique avec le mot clef **new**), soit en la redéfinissant (grâce à une déclaration à liaison dynamique avec le mot clef **override**) :

```

abstract class clA {
    public abstract Double prix { // propriété abstraite virtuelle de la classe mère
        get ; // propriété abstraite en lecture
        set ; // propriété abstraite en écriture
    }
}
class clB1 : clA {
    private Double prixTotal ;
    private Double tauxTVA = 1.196 ;

    public new Double prix { //--redéfinition par new refusée (car membre abstract)

```

```

    get { return Math.Round (prixTotal ); }
    set { prixTotal = value * tauxTVA ; }
}

class clB2 : clA {
    private Double prixTotal ;
    private Double tauxTVA = 1.05 ;
    public override Double prix { // redéfinition de la propriété par override correcte
        get { return Math.Round (prixTotal ); }
        set { prixTotal = value * tauxTVA ; }
    }
}

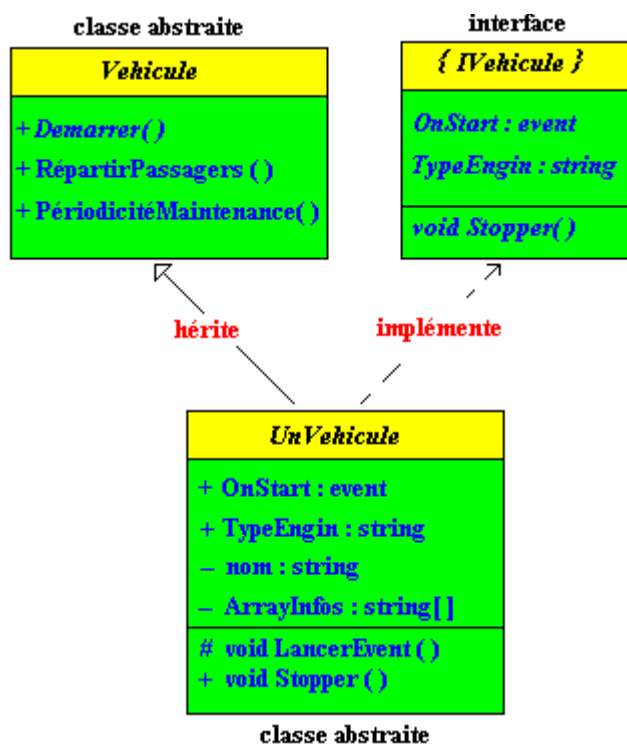
```

1.8 Les propriétés peuvent être déclarées dans une interface

Les propriétés en C# peuvent être déclarées dans une interface comme les événements et les méthodes sans le mot clef **abstract**, dans ce cas comme dans le cas de propriété abstraites la déclaration ne contient pas de corps de définition pour le ou les accesseurs qui la composent, ces accesseurs sont implémentés dans une classe fille qui implémente elle-même l'interface.

Les propriétés déclarées dans une interface lorsqu'elles sont implémentées dans une classe peuvent être définies soit à liaison statique, soit à liaison dynamique.

Ci dessous une exemple de hiérarchie abstraite de véhicules, avec une interface IVehicule contenant un événement (cet exemple est spécifié au chapitre sur les interfaces) :



```
abstract class Vehicule { // classe abstraite mère
    ....
}
```

```
interface IVehicule {
    ....
    string TypeEngin { // déclaration de propriété abstraite par défaut
        get ;
        set ;
    }
    ....
}
```

```
abstract class UnVehicule : Vehicule , IVehicule {
    private string nom = "";
    ....
    public virtual string TypeEngin { // implantation virtuelle de la propriété
        get { return nom ; }
        set { nom = "["+value+"]" ; }
    }
    ....
}
```

```
abstract class Terrestre : UnVehicule {
    ....
    public override string TypeEngin { // redéfinition de propriété
        get { return base.TypeEngin ; }
        set { string nomTerre = value + "-Terrestre";
              base.TypeEngin = nomTerre ; }
    }
}
```

1.9 Exemple complet exécutable

Code C# complet compilable avec l'événement et une classe concrète

```
public delegate void Starting (); // delegate declaration de type pour l'événement

abstract class Vehicule { // classe abstraite mère
    public abstract void Demarrer (); // méthode abstraite
    public void RépartirPassagers () { } // implantation de méthode avec corps vide
    public void PériodicitéMaintenance () { } // implantation de méthode avec corps vide
}

interface IVehicule {
    event Starting OnStart ; // déclaration d'événement du type délégué : Starting
    string TypeEngin { // déclaration de propriété abstraite par défaut
        get ;
        set ;
    }
    void Stopper (); // déclaration de méthode abstraite par défaut
}
```

```

}

//-- classe abstraite héritant de la classe mère et implémentant l'interface :
abstract class UnVehicule : Vehicule , IVehicule {
    private string nom = "";
    private string [ ] ArrayInfos = new string [10];
    public event Starting OnStart ;
    protected void LancerEvent () {
        if( OnStart != null)
            OnStart ();
    }
    public virtual string TypeEngin { // implantation virtuelle de la propriété
        get { return nom ; }
        set { nom = "["+value+"]" ; }
    }
    public virtual void Stopper () { } // implantation virtuelle de méthode avec corps vide
}

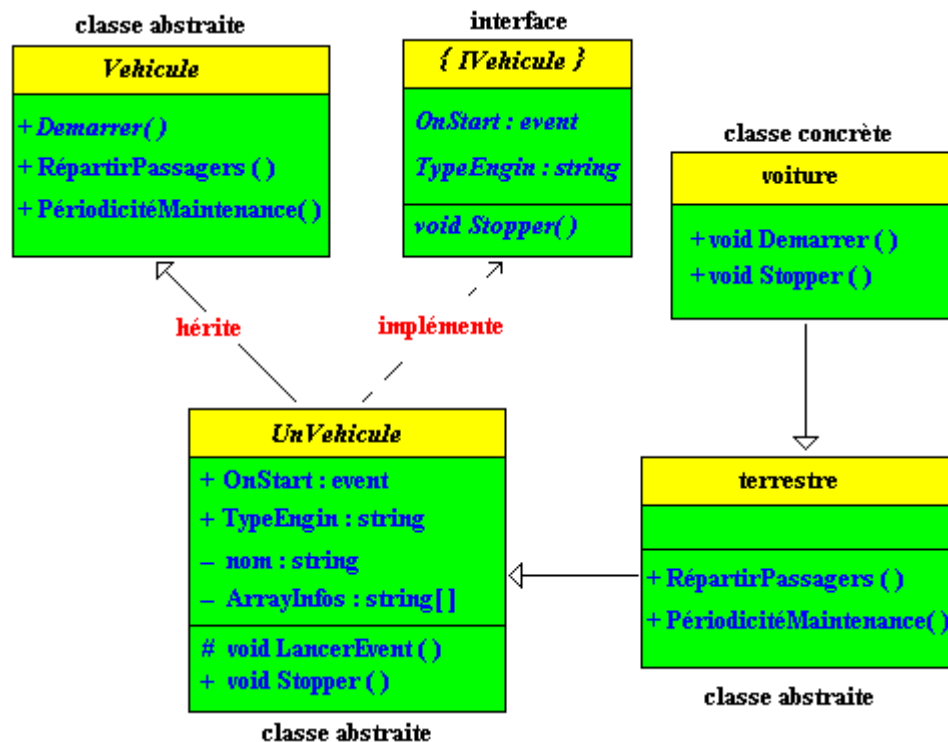
abstract class Terrestre : UnVehicule {
    private string nomTerre = "";
    public new void RépartirPassagers () { }
    public new void PériodicitéMaintenance () { }
    public override string TypeEngin { // redéfinition de propriété
        get { return base.TypeEngin ; }
        set { string nomTerre = value + "-Terrestre";
            base.TypeEngin = nomTerre ; }
    }
}

class Voiture : Terrestre {
    public override string TypeEngin { // redéfinition de propriété
        get { return base.TypeEngin + "-voiture"; }
        set { base.TypeEngin = "(" + value + ")"; }
    }
    public override void Demarrer () {
        LancerEvent();
    }
    public override void Stopper () {
        //...
    }
}

class UseVoiture { // instantiation d'une voiture particulière
    static void Main ( string [] args ) {
        UnVehicule x = new Voiture ();
        x.TypeEngin = "Picasso" ; // propriété en écriture
        System.Console.WriteLine ( "x est une " + x.TypeEngin ); // propriété en lecture
        System.Console.ReadLine ();
    }
}

```

Diagrammes de classes UML de la hiérarchie implantée :



Résultats d'exécution :

```

C:\D:\CsharpExos\propIndex\bin\Debug\ProjPropIndex.exe
x est une [<Picasso>-Terrestre]-voiture
  
```

1.9.1 Détails de fonctionnement de la propriété TypeEngin en écriture

La propriété `TypeEngin` est en écriture dans :

```
x.TypeEngin = "Picasso";
```

Elle remonte à ses définitions successives grâce l'utilisation du mot clef `base` qui fait référence à la classe mère de la classe en cours.

- propriété `TypeEngin` dans la classe `Voiture` :

```

(Picasso)
base.TypeEngin
  
```

- propriété `TypeEngin` dans la classe `Terrestre` :

```

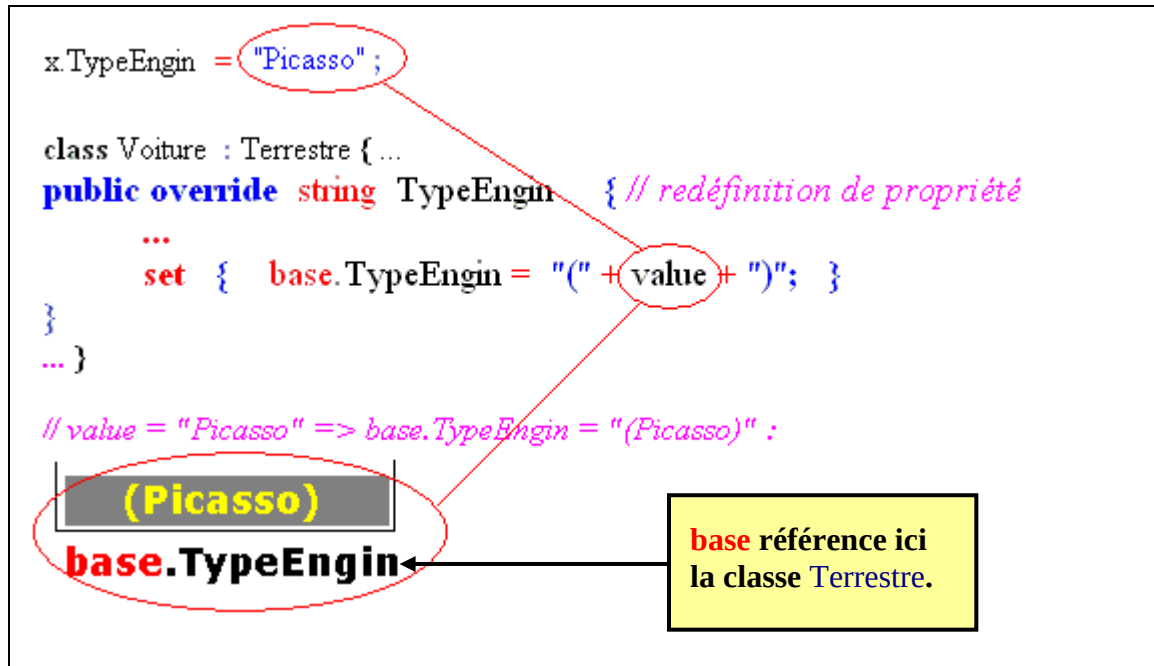
(Picasso)-Terrestre
base.TypeEngin
.....
  
```

- propriété `TypeEngin` dans la classe `UnVehicule` :

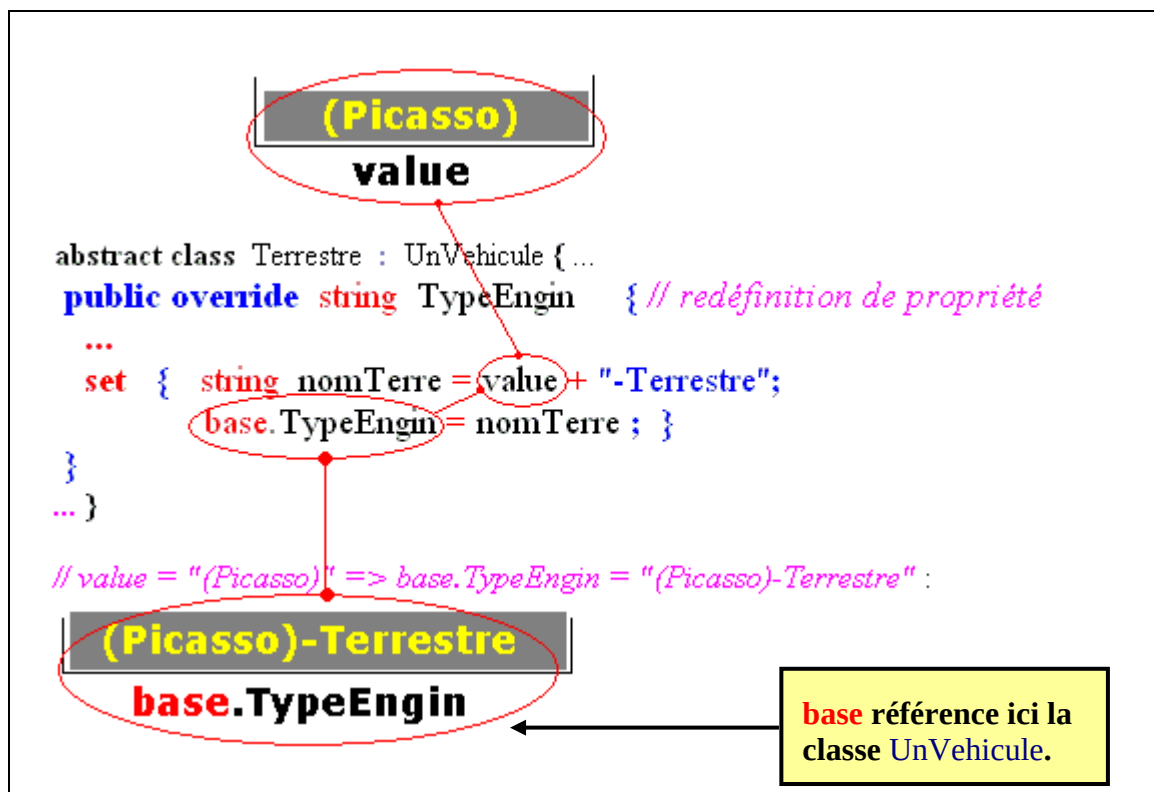
```

[(Picasso)-Terrestre]
private string nom
.....
  
```

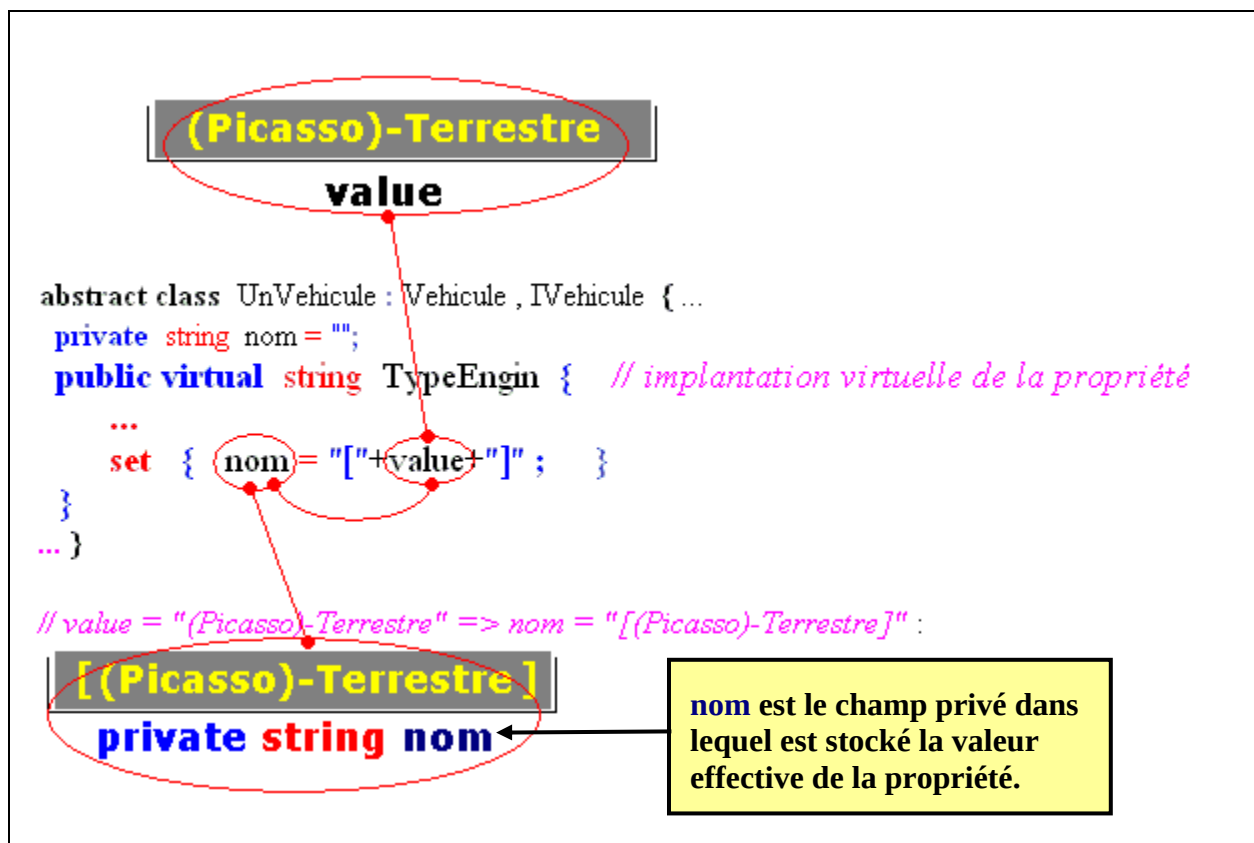
Définition de la propriété dans la classe Voiture (écriture) :



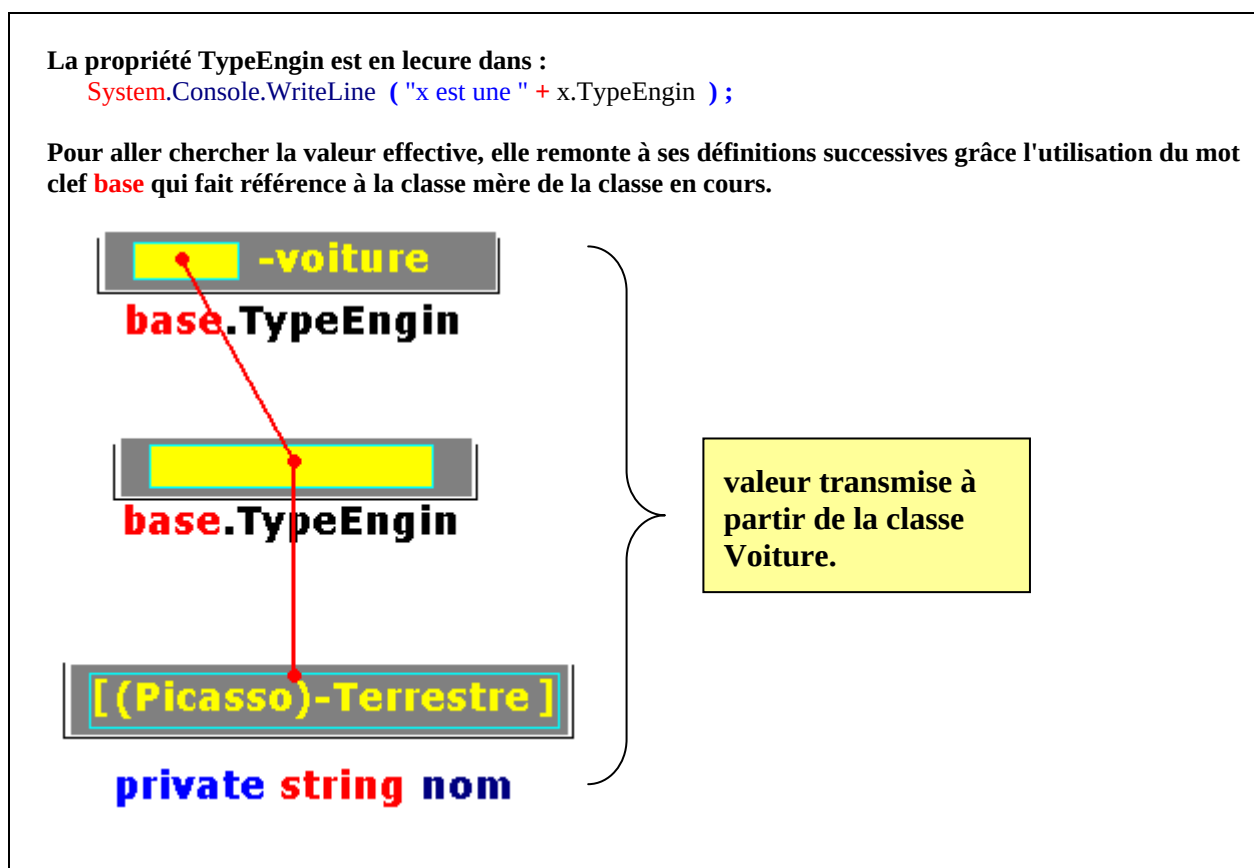
Définition de la propriété dans la classe Terrestre (écriture) :



Définition de la propriété dans la classe UnVehicule (écriture) :

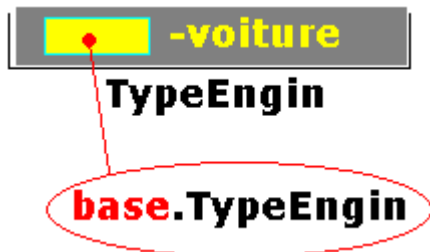


1.9.2 Détails de fonctionnement de la propriété TypeEngin en lecture



Définition de la propriété dans la classe Voiture (lecture) :

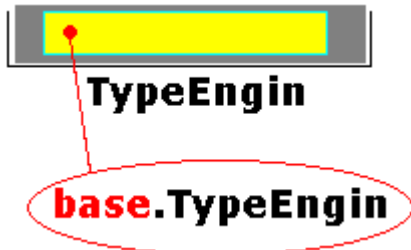
```
class Voiture : Terrestre { ...  
    public override string TypeEngin { // redéfinition de propriété  
        get { return base.TypeEngin + "-voiture"; }  
        ...  
    }  
    ...  
}
```



L'accesseur **get** va chercher le résultat dans **base.TypeEngin** et lui concatène le mot **"-voiture"**. **base.TypeEngin** référence ici la propriété dans la classe **Terrestre**.

Définition de la propriété dans la classe Terrestre (lecture) :

```
abstract class Terrestre : UnVehicule { ...  
    public override string TypeEngin { // redéfinition de propriété  
        get { return base.TypeEngin ; }  
        ...  
    }  
    ...  
}
```



L'accesseur **get** va chercher le résultat dans **base.TypeEngin**. **base.TypeEngin** référence ici la propriété dans la classe **UnVehicule**.

Définition de la propriété dans la classe UnVehicule (lecture) :

```
abstract class UnVehicule : Vehicule , IVehicule { ...  
    private string nom = "";  
    public virtual string TypeEngin { // implantation virtuelle de la propriété  
        get { return nom ; }  
        ...  
    }  
    ...  
}
```



1.10 les accesseurs *get* et *set* peuvent avoir une visibilité différente

Depuis la version 2.0, un propriété peut être "public", son accesseur d'écriture **set** (ou de lecture **get**) peut être par exemple qualifié de "**protected**" afin de ne permettre l'écriture (ou la lecture) que dans des classes filles.

Les modificateurs de visibilité des accesseurs *get* et *set* doivent être, lorsqu'ils sont présents plus restrictifs que ceux de la propriété elle-même.

Exemple de propriété sans modificateurs dans *get* et *set* :

```
class clA
{
    private int champ;
    public int prop
    {
        get { return champ; }
        set { champ = value; }
    }
}
```

Exemple de propriété avec modificateurs dans **get** et **set** :

```
class clA
{
    private int champ;
    public int prop
    {
        protected get { return champ; } //lecture dans classe fille
        protected set { champ = value; } //écriture dans classe fille
    }
}
```

2. Les indexeurs

Nous savons en Delphi qu'il existe une notion de propriété par défaut qui nous permet par exemple dans un objet `Obj` de type `TStringList` se nomme **strings**, d'écrire **Obj[5]** au lieu de **Obj.strings[5]**. La notion d'indexeur de C# est voisine de cette notion de propriété par défaut en Delphi.

2.1 Définitions et comparaisons avec les propriétés

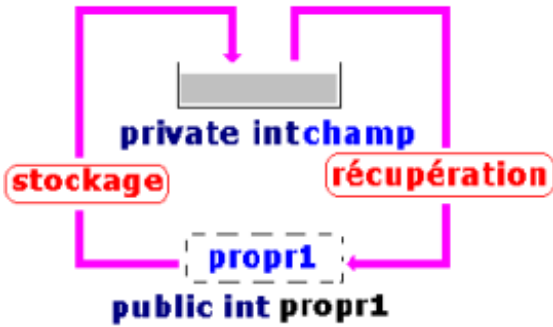
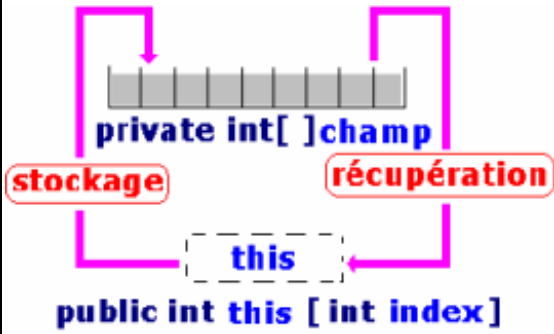
Un indexeur est un membre de classe qui permet à un objet d'être **indexé de la même manière qu'un tableau**. La signature d'un indexeur doit être différente des signatures de tous les autres indexeurs déclarés dans la même classe. Les indexeurs et les propriétés sont très similaires de par leur concept, c'est pourquoi nous allons définir les indexeurs à partir des propriétés.

Tous les indexeurs sont représentés par l'opérateur `[ ]`. Les liens sur les propriétés ou les indexeurs du tableau ci-dessous renvoient directement au paragraphe associé.

Propriété	Indexeur
Déclarée par son nom.	Déclaré par le mot clef this .
Identifiée et utilisée par son nom.	Identifié par sa signature, utilisé par l'opérateur <code>[]</code> . L'opérateur <code>[]</code> doit être situé immédiatement après le nom de l'objet.
Peut être un membre de classe static ou un membre d'instance.	Ne peut pas être un membre static , est toujours un membre d'instance.
L'accesseur get correspond à une méthode sans paramètre.	L'accesseur get correspond à une méthode pourvue de la même liste de paramètres formels que l'indexeur.
L'accesseur set correspond à une méthode avec un seul paramètre implicite value .	L'accesseur set correspond à une méthode pourvue de la même liste de paramètres formels que l'indexeur plus le paramètre implicite value .
Une propriété <code>Prop</code> héritée est accessible par la syntaxe base.Prop	Un indexeur <code>Prop</code> hérité est accessible par la syntaxe base.[]
Les propriétés peuvent être à liaison statique, à liaison dynamique, masquées ou redéfinies.	Les indexeurs peuvent être à liaison statique, à liaison dynamique, masqués ou redéfinis.

Les propriétés peuvent être abstraites.	Les indexeurs peuvent être abstraits.
Les propriétés peuvent être déclarées dans une interface.	Les indexeurs peuvent être déclarés dans une interface.

2.1.1 Déclaration

Propriété	Indexeur
Déclarée par son nom, avec champ de stockage : <pre>private int champ;</pre> <pre>public int propr1{ get { return champ ; } set { champ = value ; } }</pre> 	Déclaré par le mot clef this, avec champ de stockage : <pre>private int [] champ = new int [10];</pre> <pre>public int this [int index]{ get { return champ[index] ; } set { champ[index] = value ; } }</pre> 

2.1.2 Utilisation

Propriété	Indexeur
<u>Déclaration :</u> <pre>class clA { private int champ;</pre> <pre> public int propr1{ get { return champ ; } set { champ = value ; } } }</pre> <u>Utilisation :</u> <pre>clA Obj = new clA();</pre>	<u>Déclaration :</u> <pre>class clA { private int [] champ = new int [10];</pre> <pre> public int this [int index]{ get { return champ[index] ; } set { champ[index] = value ; } } }</pre> <u>Utilisation :</u> <pre>clA Obj = new clA();</pre>

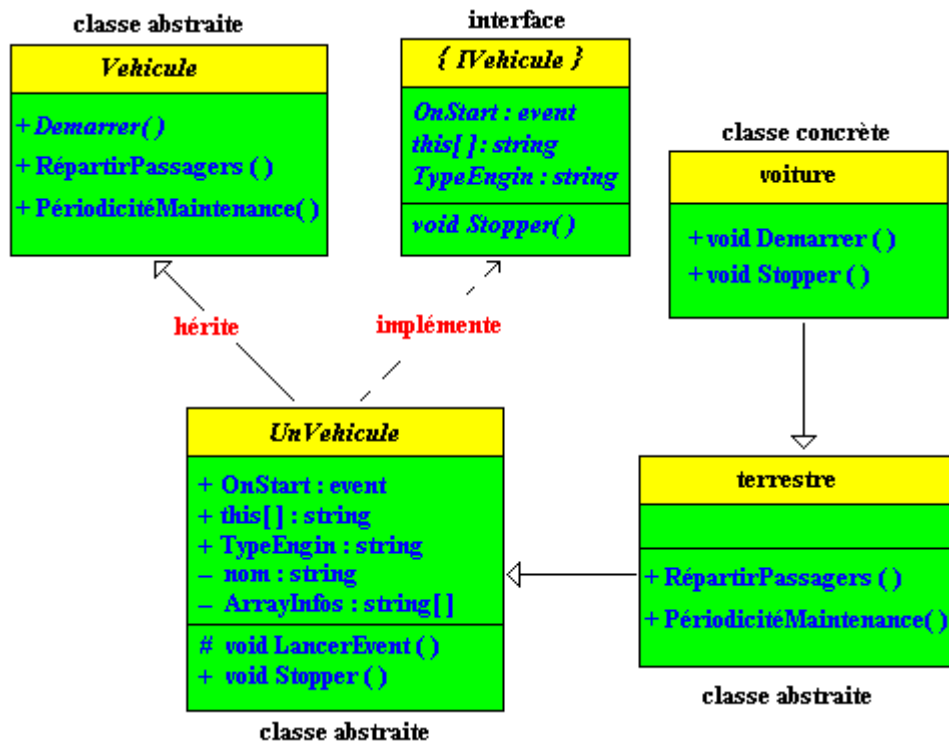
Obj.propr1 = 100 ; private int champ stockage recupération propr1 public int propr1 int x = Obj.prop1 ; // x = 100	for (int i =0; i<5; i++) Obj[i] = i ; private int[] champ stockage recupération this public int this [int index] int x = Obj[2] ; // x = 2 int y = Obj[3] ; // x = 3 ...
---	---

2.1.3 Paramètres

Propriété	Indexeur
<u>Déclaration :</u> class clA { private int champ; public int propr1{ get { return champ*10 ; } set { champ = value + 1 ; } } } value est un paramètre implicite. <u>Utilisation :</u> clA Obj = new clA(); Obj.propr1 = 100 ; int x = Obj.prop1 ; // x = 1010	<u>Déclaration :</u> class clA { private int [] champ = new int [10]; public int this [int k]{ get { return champ[k]*10 ; } set { champ[k] = value + 1 ; } } } k est un paramètre formel de l'indexeur. <u>Utilisation :</u> clA Obj = new clA(); for (int i =0; i<5; i++) Obj[i] = i ; int x = Obj[2] ; // x = 30 int y = Obj[3] ; // x = 40 ...

2.1.4 Indexeur à liaison dynamique, abstraction et interface

Reprenons l'exemple de hiérarchie de véhicules, traité avec la propriété TypeEngin de type string, en y ajoutant un indexeur de type string en proposant des définitions parallèles à la propriété et à l'indexeur :



```

abstract class Vehicule { // classe abstraite mère
    ....
}

```

```

interface IVehicule {
    ....
    string TypeEngin { // déclaration de propriété abstraite par défaut
        get ;
        set ;
    }
    ....
    string this [ int k ] { // déclaration d'indexeur abstrait par défaut
        get ;
        set ;
    }
    ....
}

```

```

abstract class UnVehicule : Vehicule , IVehicule {
    private string [ ] ArrayInfos = new string [10] ;
    private string nom = "" ;
    ....
    public virtual string TypeEngin { // implantation virtuelle de la propriété
        get { return nom ; }
        set { nom = "["+value+"]" ; }
    }
    ....
    public virtual string this [ int k ] { // implantation virtuelle de l'indexeur
        get { return ArrayInfos[ k ] ; }
        set { ArrayInfos[ k ] = value ; }
    }
}

```

```

    }
    ....
}

```

```

abstract class Terrestre : UnVehicule {
    private string nomTerre = "";
    ....
    public override string TypeEngin { // redéfinition de propriété
        get { return base.TypeEngin ; }
        set { string nomTerre = value + "-Terrestre";
            base.TypeEngin = nomTerre ; }
    }
    ....
    public override string this [ int k ] { // redéfinition de l'indexeur
        get { return base[ k ] ; }
        set { string nomTerre = value + "-Terrestre" ;
            base[ k ] = nomTerre + "/set =" + k.ToString() + "/" ; }
    }
}

```

```

class Voiture : Terrestre {
    public override string TypeEngin { // redéfinition de propriété
        get { return base.TypeEngin + "-voiture" ; }
        set { base.TypeEngin = "(" + value + ")"; }
    }
    public override string this [ int n ] { // redéfinition de l'indexeur
        get { return base[ n ] + "-voiture{get =" + n.ToString() + "}" ; }
        set { base[ n ] = "(" + value + ")"; }
    }
    ...
}

```

Code C# complet compilable

Code avec un événement une propriété et un indexeur

```

public delegate void Starting ( ); // delegate declaration de type

abstract class Vehicule {
    public abstract void Demarrer ( );
    public void RépartirPassagers ( ) { }
    public void PériodicitéMaintenance ( ) { }
}

interface IVehicule {
    event Starting OnStart ; // déclaration événement
    string this [ int index] // déclaration Indexeur
    {
        get ;
        set ;
    }
}

```

```

string TypeEngin // déclaration propriété
{
    get ;
    set ;
}
void Stopper ();
}

abstract class UnVehicule : Vehicule, IVehicule {
    private string nom = "";
    private string [] ArrayInfos = new string [10] ;
    public event Starting OnStart ; // implantation événement
    protected void LancerEvent () {
        if( OnStart != null)
            OnStart ();
    }
    public virtual string this [ int index] { // implantation indexeur virtuel
        get { return ArrayInfos[index] ; }
        set { ArrayInfos[index] = value ; }
    }
    public virtual string TypeEngin { // implantation propriété virtuelle
        get { return nom ; }
        set { nom = "[" + value + "]"; }
    }
    public virtual void Stopper () { }
}

abstract class Terrestre : UnVehicule {
    public new void RépartirPassagers () { }
    public new void PériodicitéMaintenance () { }
    public override string this [ int k] { // redéfinition indexeur
        get { return base [k] ; }
        set { string nomTerre = value + "-Terrestre";
            base [k] = nomTerre + "/set = " + k.ToString () + "/";
        }
    }
    public override string TypeEngin { // redéfinition propriété
        get { return base .TypeEngin ; }
        set { string nomTerre = value + "-Terrestre";
            base .TypeEngin = nomTerre ;
        }
    }
}

class Voiture : Terrestre {
    public override string this [ int n] { // redéfinition indexeur
        get { return base [n] + "-voiture {get = " + n.ToString () + " }"; }
        set { string nomTerre = value + "-Terrestre";
            base [n] = "(" + value + ")";
        }
    }
    public override string TypeEngin { // redéfinition propriété
        get { return base .TypeEngin + "-voiture"; }
        set { base .TypeEngin = "(" + value + ")"; }
    }
    public override void Demarrer () {
        LancerEvent ();
    }
    public override void Stopper () {
        //...
    }
}

```

```

}

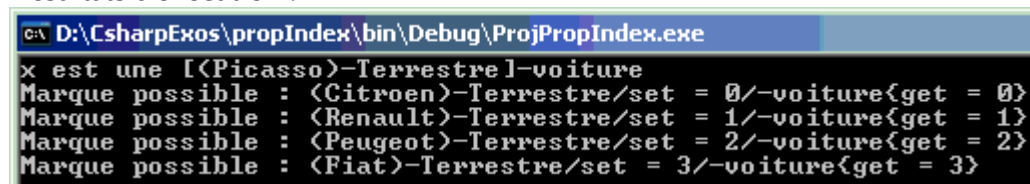
class UseVoiture
{
    static void Main ( string [] args )
    {
        // instantiation d'une voiture particulière :
        UnVehicule automobile = new Voiture ();

        // utilisation de la propriété TypeEngin :
        automobile.TypeEngin = "Picasso";
        System.Console.WriteLine ("x est une " + automobile.TypeEngin );

        // utilisation de l'indexeur :
        automobile [0] = "Citroen";
        automobile [1] = "Renault";
        automobile [2] = "Peugeot";
        automobile [3] = "Fiat";
        for( int i = 0 ; i < 4 ; i ++ )
            System.Console.WriteLine ("Marque possible : " + automobile [i] );
        System.Console.ReadLine ();
    }
}

```

Résultats d'exécution :



```

C:\> D:\CsharpExos\propIndex\bin\Debug\ProjPropIndex.exe
x est une [<Picasso>-Terrestre]-voiture
Marque possible : <Citroen>-Terrestre/set = 0/-voiture<get = 0>
Marque possible : <Renault>-Terrestre/set = 1/-voiture<get = 1>
Marque possible : <Peugeot>-Terrestre/set = 2/-voiture<get = 2>
Marque possible : <Fiat>-Terrestre/set = 3/-voiture<get = 3>

```

Application fenêtrées et ressources en



Plan général:

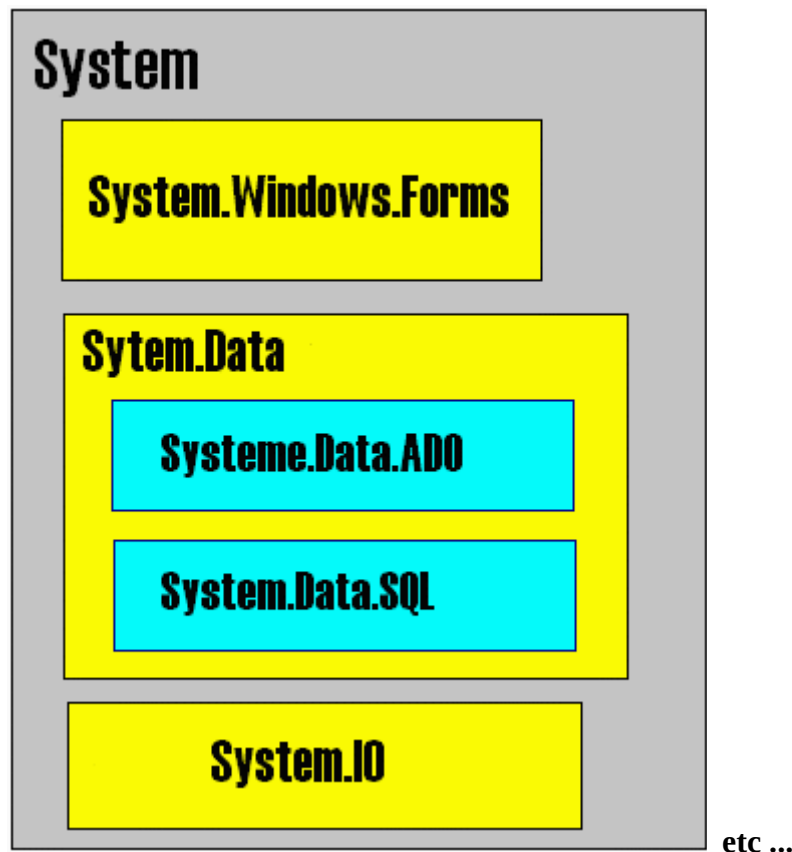
1. Les applications avec Interface Homme-Machine

- 1.1 Un retour sur la console
- 1.2 Des fenêtres à la console
 - 1.2.1 Console et fenêtre personnalisée
 - 1.2.2 Fenêtre personnalisée sans console
 - 1.2.3 Que fait Application.Run ?
 - 1.2.4 Que faire avec Application.DoEvents ?
- 1.3 Un formulaire en C# est une fiche
- 1.4 Code C# engendré par le RAD pour un formulaire
- 1.5 Libération de ressources non managées
- 1.6 Comment la libération a-t-elle lieu dans le NetFrameWork ?
- 1.7 Peut-on influencer sur cette la libération dans le NetFrameWork ?
- 1.8 Design Pattern de libération des ressources non managées
- 1.9 Un exemple utilisant la méthode Dispose d'un formulaire
- 1.10 L'instruction USING appelle Dispose()
- 1.11 L'attribut [STAThread]

1. Les applications avec Interface Homme-Machine

Les exemples et les traitements qui suivent sont effectués sous l'OS windows depuis la version NetFrameWork 1.1, les paragraphes 1.3, 1.4, ... , 1.10 expliquent le contenu du code généré automatiquement par Visual Studio ou C# Builder de Borland Studio pour développer une application fenêtrée.

Le NetFrameWork est subdivisé en plusieurs espaces de nom, l'espace de noms System contient plusieurs classes, il est subdivisé lui-même en plusieurs sous-espaces de noms :



L'espace des noms **System.Windows.Forms** est le domaine privilégié du NetFrameWork dans lequel l'on trouve des classes permettant de travailler sur des applications fenêtrées.

La classe **Form** de l'espace des noms **System.Windows.Forms** permet de créer une fenêtre classique avec barre de titre, zone client, boutons de fermeture, de zoom...

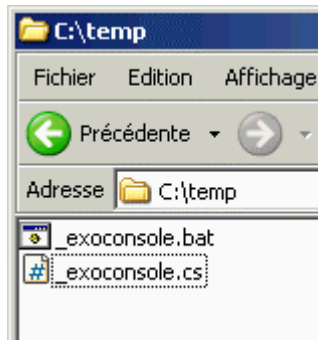
En C#, Il suffit d'instancier un objet de cette classe pour obtenir une fenêtre classique qui s'affiche sur l'écran.

1.1 un retour sur la console

Les exemples sont écrits pour la version 1.1 mais restent valides pour les versions ultérieures.

Le code C# peut tout comme le code Java être écrit avec un éditeur de texte rudimentaire du style bloc-note, puis être compilé directement à la **console** par appel au compilateur **csc.exe**.

Soient par exemple dans un dossier temp du disque C: , deux fichiers :



Le fichier "**_exoconsole.bat**" contient la commande système permettant d'appeler le compilateur C#.

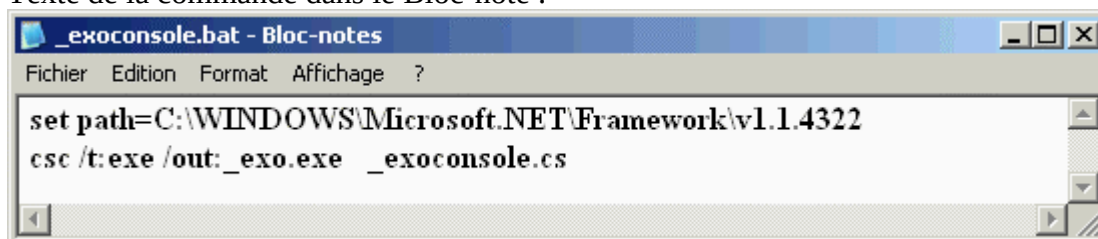
Le fichier "**_exoconsole.cs**" le programme source en C#.

Construction de la commande de compilation "**_exoconsole.bat**" :

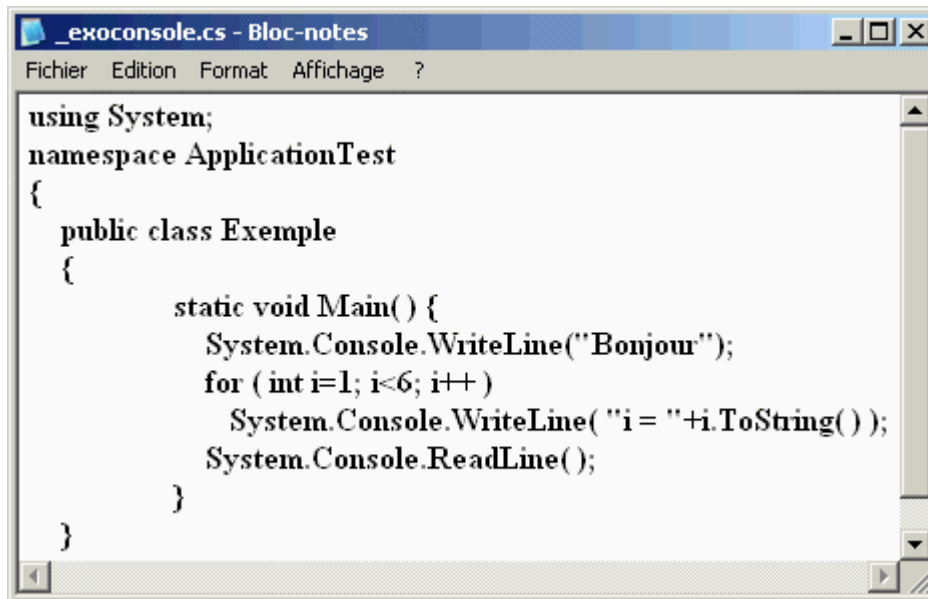
On indique d'abord le chemin (variable path) du répertoire où se trouve le compilateur **csc.exe**, puis on lance l'appel au compilateur avec ici , un nombre minimal de paramètres :

Attributs et paramètres de la commande	fonction associée
set path = C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322	Le chemin absolu permettant d'accéder au dossier contenant le compilateur C# (csc.exe)
/t:exe	Indique que nous voulons engendrer une exécutable console (du code MSIL)
/out: _exo.exe	Indique le nom que doit porter le fichier exécutable MSIL après compilation
_exoconsole.cs	Le chemin complet du fichier source C# à compiler (ici il est dans le même répertoire que la commande, seul le nom du fichier suffit)

Texte de la commande dans le Bloc-note :



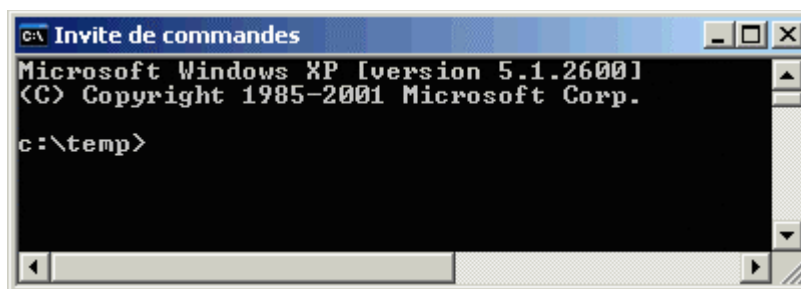
Nous donnons le contenu du fichier source **_exoconsole.cs** à compiler :



```
using System;
namespace ApplicationTest
{
    public class Exemple
    {
        static void Main() {
            System.Console.WriteLine("Bonjour");
            for (int i=1; i<6; i++)
                System.Console.WriteLine( "i = "+i.ToString() );
            System.Console.ReadLine();
        }
    }
}
```

Le programme précédent affiche le mot Bonjour suivi de l'exécution de la boucle sur 5 itérations.

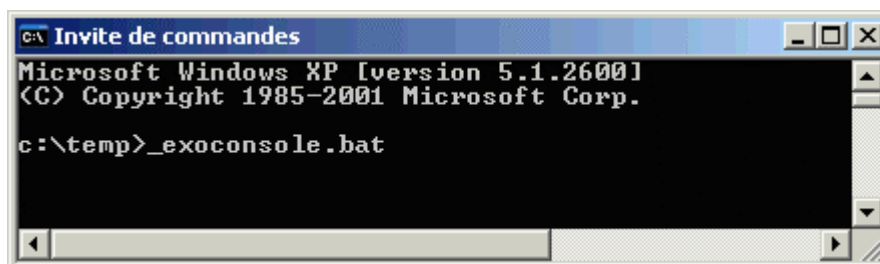
On lance l'invite de commande de Windows ici dans le répertoire c:\temp :



```
C:\> Invite de commandes
Microsoft Windows XP [version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\temp>
```

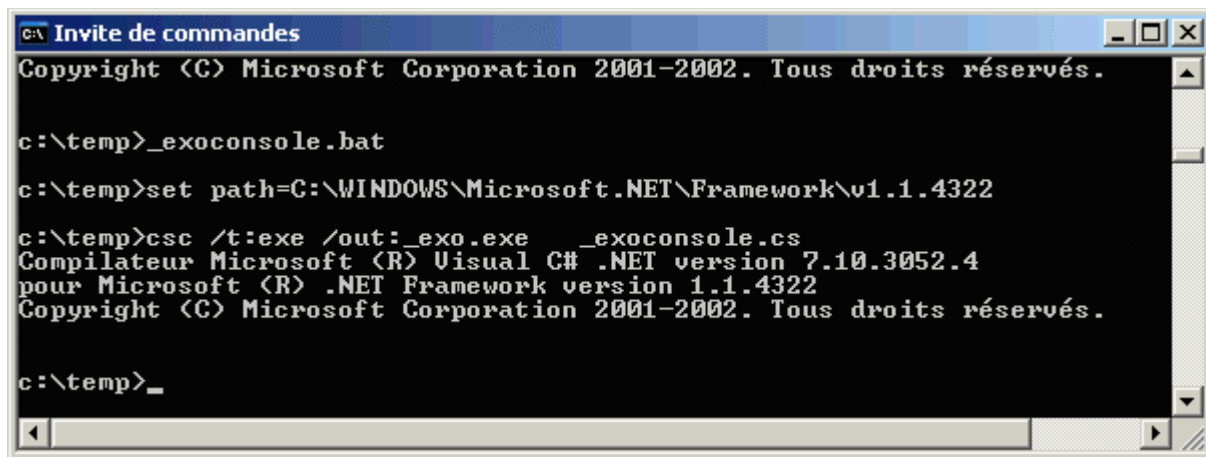
On tape au clavier et l'on exécute la commande "**_exoconsole.bat**" :



```
C:\> Invite de commandes
Microsoft Windows XP [version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\temp>_exoconsole.bat
```

Elle appelle le compilateur **csc.exe** qui effectue la compilation du programme **_exoconsole.cs** sans signaler d'erreur particulière et qui a donc engendré un fichier MSIL nommé **_exo.exe** :



```
C:\> Invite de commandes
Copyright (C) Microsoft Corporation 2001-2002. Tous droits réservés.

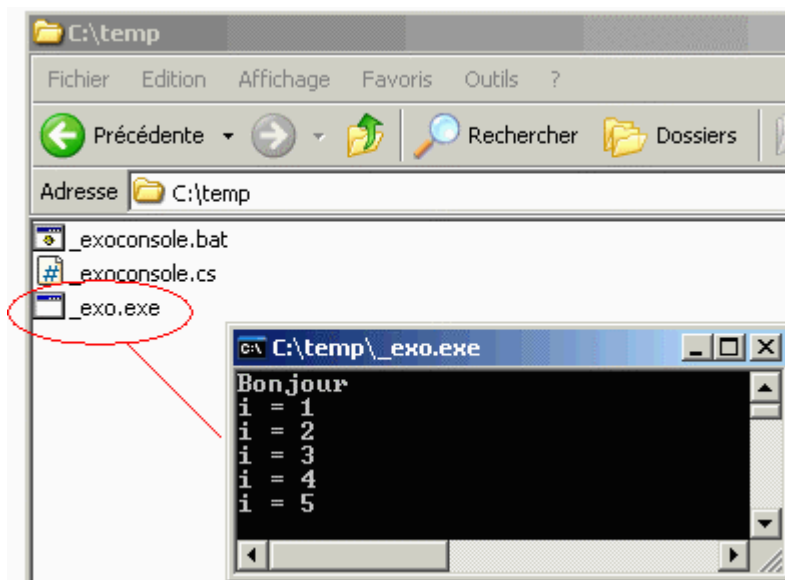
c:\temp>_exoconsole.bat

c:\temp>set path=C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322

c:\temp>csc /t:exe /out:_exo.exe _exoconsole.cs
Compilateur Microsoft (R) Visual C# .NET version 7.10.3052.4
pour Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. Tous droits réservés.

c:\temp>_
```

Lançons par un double click l'exécution du programme **_exo.exe** qui vient d'être engendré :

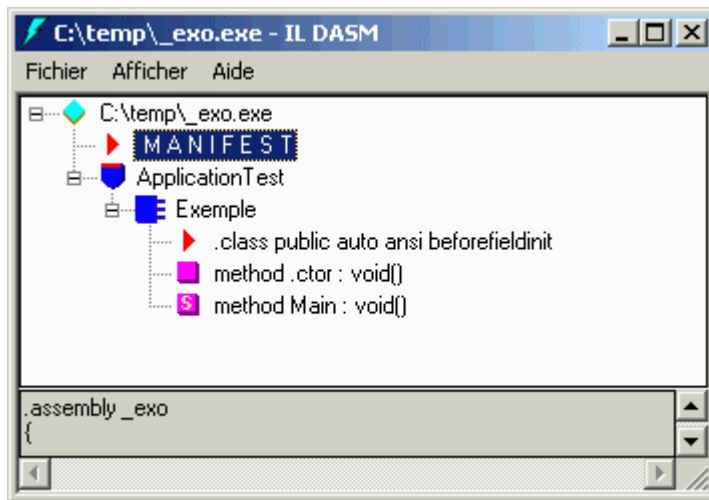


Nous constatons que l'exécution par le CLR du fichier **_exo.exe** a produit le résultat escompté c'est à dire l'affichage du mot Bonjour suivi de l'exécution de la boucle sur 5 itérations.

Afin que le lecteur soit bien convaincu que nous sommes sous NetFramework et que les fichiers exécutables ne sont pas du binaire exécutable comme jusqu'à présent sous Windows, mais des fichiers de code MSIL exécutable par le CLR, nous passons le fichier **_exo.exe** au désassembleur **ildasm** par la commande "ildasm.bat".

Le désassembleur MSIL Disassembler (Ildasm.exe) est un utilitaire inclus dans le kit de développement .NET Framework SDK, il est de ce fait utilisable avec tout langage de .Net dont C#. ILDasm.exe analyse toutes sortes d'assemblys .NET Framework **.exe** ou **.dll** et présente les informations dans un format explicite. Cet outil affiche bien plus que du code MSIL (Microsoft Intermediate Language) ; il présente également les espaces de noms et les types, interfaces comprises.

Voici l'inspection du fichier `_exo.exe` par **ildasm** :



Demandons à **ildasm** l'inspection du code MSIL engendré pour la méthode `Main()` :

Nous avons mis en **gras et en italique** les commentaires d'instructions sources

Exemple::methodeMain void()

```
.method private hidebysig static void Main( ) cil managed
{
    .entrypoint
    // Code size      51 (0x33)
    .maxstack 2
    .locals init ([0] int32 i)
    .language '{3F5162F8-07C6-11D3-9053-00C04FA302A1}', '{994B45C4-E6E9-11D2-903F-00C04FA302A1}', '{5A869D0B-6611-11D3-BD2A-0000F80849BD}'
    // Source File 'c:\temp\_exoconsole.cs'
    //000007:      System.Console.WriteLine("Bonjour");
    IL_0000: ldstr    "Bonjour"
    IL_0005: call     void [mscorlib]System.Console::WriteLine(string)
    //000008:      for ( int i=1; i<6; i++ )
    IL_000a: ldc.i4.1
    IL_000b: stloc.0
    IL_000c: br.s     IL_0028
    //000009:      System.Console.WriteLine( "i = "+i.ToString() );
    IL_000e: ldstr    "i = "
    IL_0013: ldloc.s  i
    IL_0015: call     instance string [mscorlib]System.Int32::ToString()
    IL_001a: call     string [mscorlib]System.String::Concat(string,string)
    IL_001f: call     void [mscorlib]System.Console::WriteLine(string)
    //000008:      for ( int i=1; i<6; i++ )
    IL_0024: ldloc.0
    IL_0025: ldc.i4.1
    IL_0026: add
    IL_0027: stloc.0
    IL_0028: ldloc.0
    IL_0029: ldc.i4.6
    IL_002a: blt.s   IL_000e
    //000009:      System.Console.WriteLine( "i = "+i.ToString() );
    //000010:      System.Console.ReadLine();
    IL_002c: call     string [mscorlib]System.Console::ReadLine()
    IL_0031: pop
```

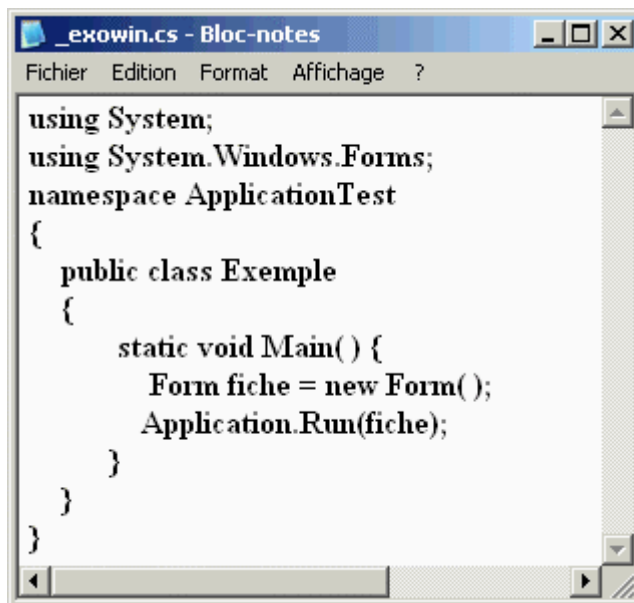
```
//000011:  }  
IL_0032: ret  
} // end of method Exemple::Main
```

1.2 Des fenêtres à la console

On peut donc de la même façon compiler et exécuter à partir de la console, des programmes C# contenant des fenêtres, comme en java il est possible d'exécuter à partir de la console des applications contenant des Awt ou des Swing, idem en Delphi. Nous proposons au lecteur de savoir utiliser des programmes qui allient la console à une fenêtre classique, ou des programmes qui ne sont formés que d'une fenêtre classique (à minima). Ici aussi, les exemples sont écrits en version 1.1 mais sont valides pour les versions ultérieures.

1.2.1 fenêtre console et fenêtre personnalisée ensembles

Ecrivons le programme C# suivant dans un fichier que nous nommons "**_exowin.cs**" :



Ce programme "**_exowin.cs**" utilise la classe Form et affiche une fenêtre de type Form :

La première instruction instancie un objet nommé **fiche** de la classe **Form**
Form fiche = new Form() ;

La fenêtre **fiche** est ensuite "initialisée" et "lancée" par l'instruction
Application.Run(fiche) ;

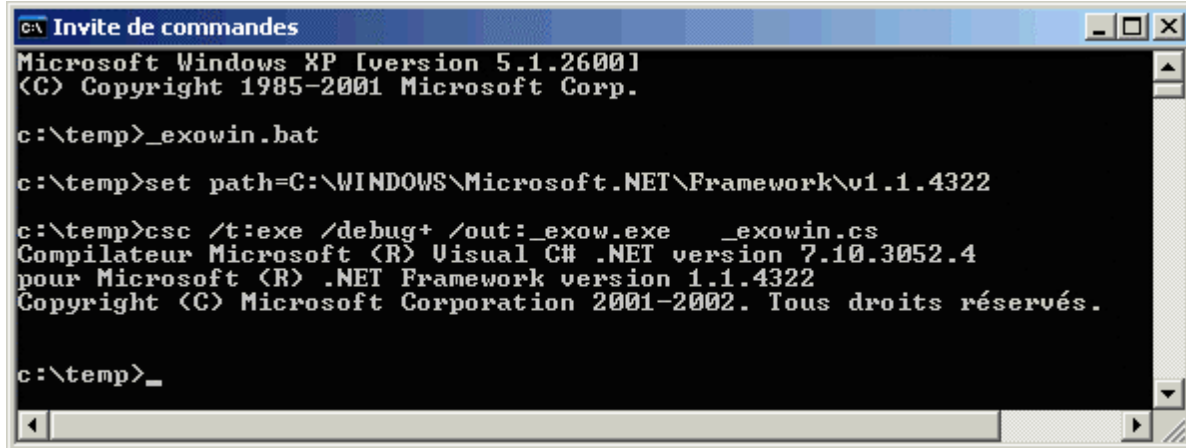
On construit une commande nommée **_exowin.bat**, identique à celle du paragraphe précédent, afin de lancer la compilation du programme **_exowin.cs**, nous décidons de nommer **_exow.exe**

l'exécutable MSIL obtenu après compilation.

contenu fichier de commande **_exowin.bat** :

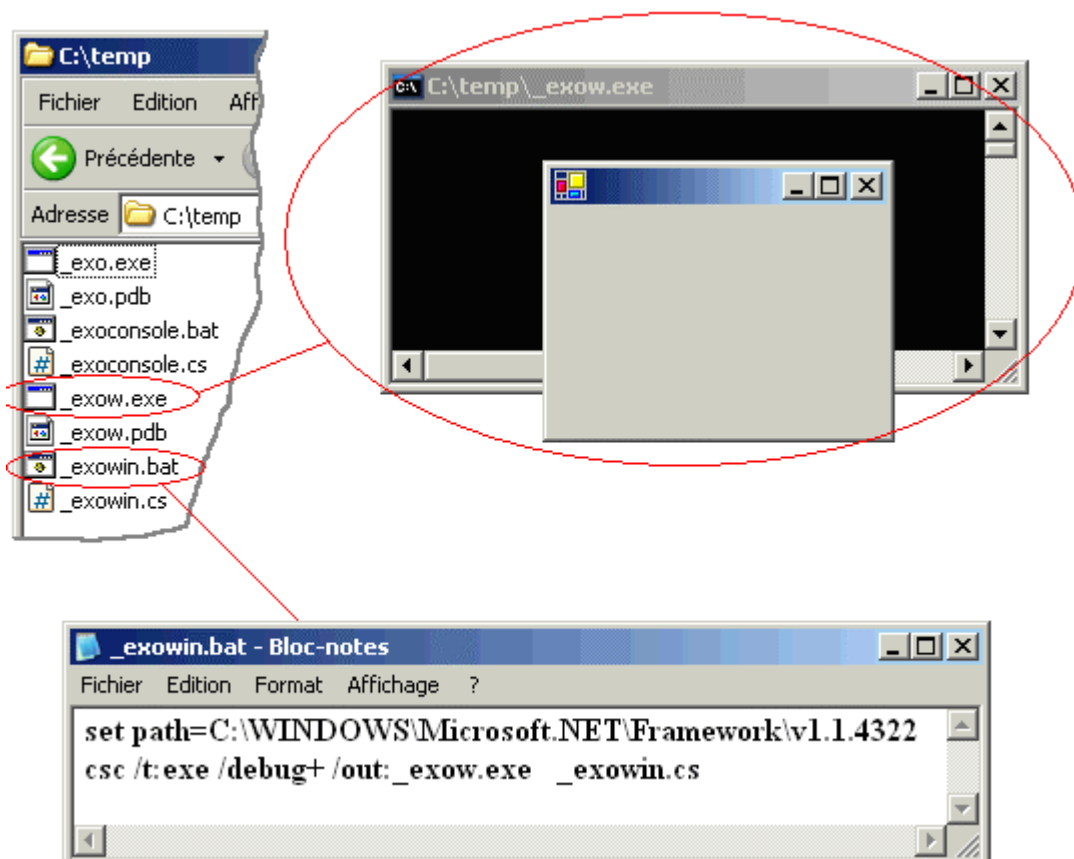
```
set path=C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
csc /t:exe /out:_exow.exe _exowin.cs
```

On exécute ensuite la commande **_exowin.bat** dans une invite de commande de Windows :



```
c:\temp>_exowin.bat
c:\temp>set path=C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
c:\temp>csc /t:exe /out:_exow.exe _exowin.cs
Compilateur Microsoft (R) Visual C# .NET version 7.10.3052.4
pour Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. Tous droits réservés.
c:\temp>
```

Cette commande a généré comme précédemment l'exécutable MSIL nommé ici **_exow.exe** dans le dossier `c:\temp`, nous exécutons le programme **_exow.exe** et nous obtenons l'affichage d'une fenêtre de console et de la fenêtre fiche :



Remarque:

L'attribut **/debug+** rajouté ici, permet d'engendrer un fichier **_exo.pdb** qui contient des informations de déboguage utiles à ildasm.

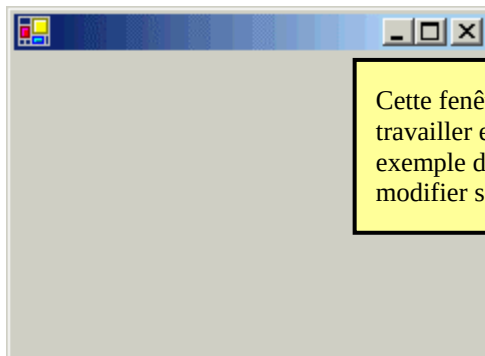
1.2.2 fenêtre personnalisée sans fenêtre console

Si nous ne voulons pas voir apparaître de fenêtre de console mais seulement la fenêtre fiche, il faut alors changer dans le paramétrage du compilateur l'attribut **target**. De la valeur **csc /t:exe** il faut passer à la valeur **csc /t:winexe** :

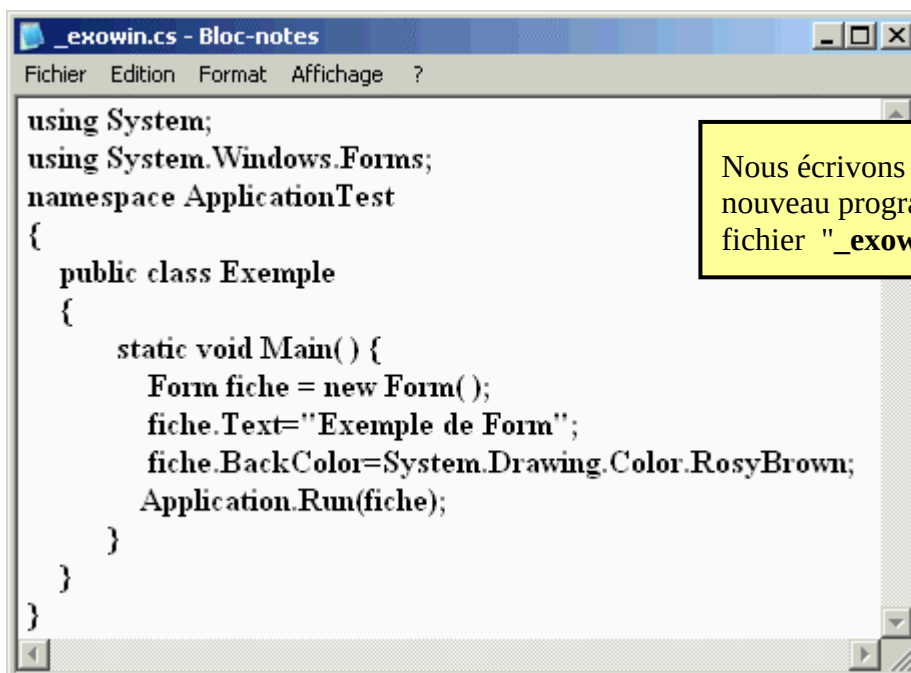
Nouveau fichier de commande **_exowin.bat** :

```
set path=C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322  
csc /t:winexe /out:_exow.exe _exowin.cs
```

Nous compilons en lançant la nouvelle commande **_exowin.bat** puis nous exécutons le nouveau programme **_exow.exe**. Nous obtenons cette fois-ci l'affichage d'une seule fenêtre (la fenêtre de console a disparu) :

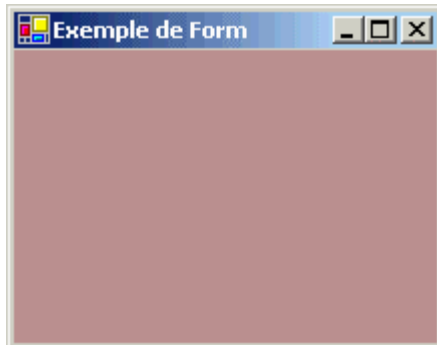


Cette fenêtre est un peu trop terne, nous pouvons travailler en mode console sur la fiche Form, afin par exemple de mettre un texte dans la barre de titre et de modifier sa couleur de fond.



Nous écrivons pour cela le nouveau programme C# dans le fichier **"_exowin.cs"**

Nous compilons en lançant la commande **_exowin.bat** puis nous exécutons le nouveau programme **_exow.exe** et nous obtenons l'affichage de la fenêtre fiche avec le texte "Exemple de Form" dans la barre de titre et sa couleur de fond marron-rosé (Color.RosyBrown) :



Consultons à titre informatif avec **ildasm** le code MSIL engendré pour la méthode Main() :

*Nous avons mis en **gras et en italique** les commentaires d'instructions sources*

Exemple::methodeMain void()

```
.method private hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size      35 (0x23)
    .maxstack 2
    .locals init ([0] class [System.Windows.Forms]System.Windows.Forms.Form fiche)
    .language '{3F5162F8-07C6-11D3-9053-00C04FA302A1}', '{994B45C4-E6E9-11D2-903F-00C04FA302A1}',
    '{5A869D0B-6611-11D3-BD2A-0000F80849BD}'
    // Source File 'c:\temp\_exowin.cs'
    //000008:      Form fiche = new Form();
    IL_0000: newobj     instance void [System.Windows.Forms]System.Windows.Forms.Form::.ctor()
    IL_0005: stloc.0
    //000009:      fiche.Text="Exemple de Form";
    IL_0006: ldloc.0
    IL_0007: ldstr      "Exemple de Form"
    IL_000c: callvirt     instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Text(string)
    //000010:      fiche.BackColor=System.Drawing.Color.RosyBrown;
    IL_0011: ldloc.0
    IL_0012: call        valuetype [System.Drawing]System.Drawing.Color
    [System.Drawing]System.Drawing.Color::get_RosyBrown()
    IL_0017: callvirt     instance void
    [System.Windows.Forms]System.Windows.Forms.Control::set_BackColor(valuetype
    [System.Drawing]System.Drawing.Color)
    //000011:      Application.Run(fiche);
    IL_001c: ldloc.0
    IL_001d: call        void [System.Windows.Forms]System.Windows.Forms.Application::Run(class
    [System.Windows.Forms]System.Windows.Forms.Form)
    //000012:      }
    IL_0022: ret
} // end of method Exemple::Main
```

1.2.3 Que fait Application.Run(fiche)

Comme les fenêtres dans Windows ne sont pas des objets ordinaires, pour qu'elles fonctionnent correctement vis à vis des messages échangés entre la fenêtre et le système, il est nécessaire de lancer une boucle d'attente de messages du genre :

```
tantque non ArrêtSysteme faire
  si événement alors
    construire Message ;
    si Message ≠ ArrêtSysteme alors
      reconnaître la fenêtre à qui est destinée ce Message;
      distribuer ce Message
    fsi
  fsi
ftant
```

La documentation technique indique que l'une des surcharges de la méthode de classe **Run** de la classe Application "**public static void Run(Form mainForm);**" *exécute une boucle de messages d'application standard sur le thread en cours et affiche la Form spécifiée*. Nous en déduisons que notre fenêtre fiche est bien initialisée et affichée par cette méthode Run.

Observons à contrario ce qui se passe si nous n'invoquons pas la méthode **Run** et programmons l'affichage de la fiche avec sa méthode **Show** :

```
using System;
using System.Windows.Forms;
namespace ApplicationTest
{
    public class Exemple
    {
        static void Main() {
            Form fiche = new Form( );
            fiche.Text="Exemple de Form";
            fiche.BackColor=System.Drawing.Color.RosyBrown;
            fiche.Show( );
        }
    }
}
```

Lorsque nous compilons puis exécutons ce programme la fiche apparaît correctement (titre et couleur) d'une manière fugace car elle disparaît aussi vite qu'elle est apparue. En effet le programme que nous avons écrit est correct :

Ligne d'instruction du programme	Que fait le CLR
{	il initialise l'exécution de la méthode Main
Form fiche = new Form();	il instancie une Form nommée fiche
fiche.Text="Exemple de Form";	il met un texte dans le titre de fiche
fiche.BackColor=System.Drawing.Color.RosyBrown;	il modifie la couleur du fond de fiche

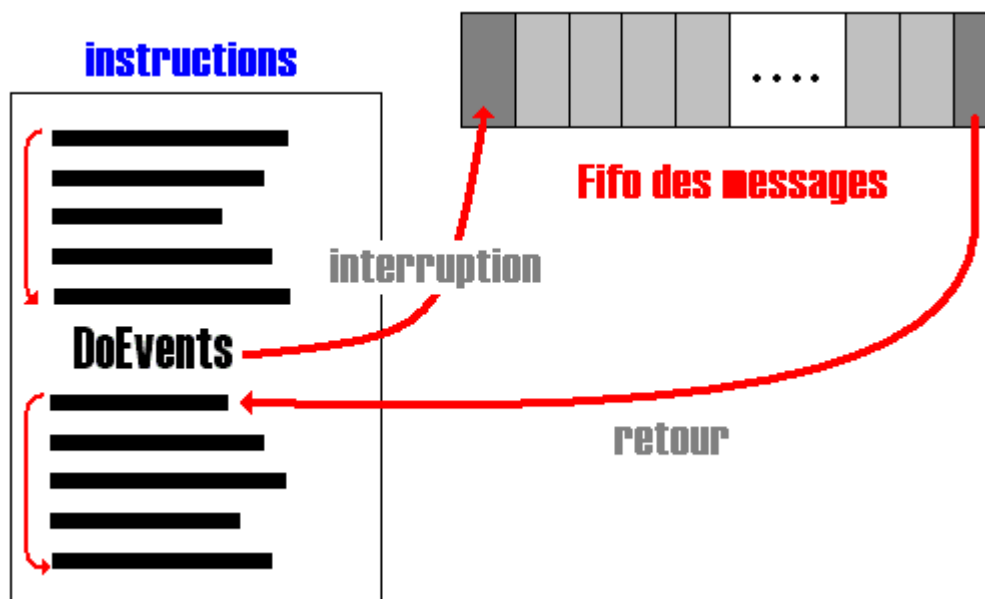
fiche.Show();	il rend la fiche visible
}	fin de la méthode Main et donc tout est détruit et libéré et le processus est terminé.

La fugacité de l'affichage de notre fenêtre fiche est donc normale, puisqu'à peine créée la fiche a été détruite.

Si nous voulons que notre objet de fiche persiste sur l'écran, il faut simuler le comportement de la méthode classe **Run**, c'est à dire qu'il nous faut écrire une boucle de messages.

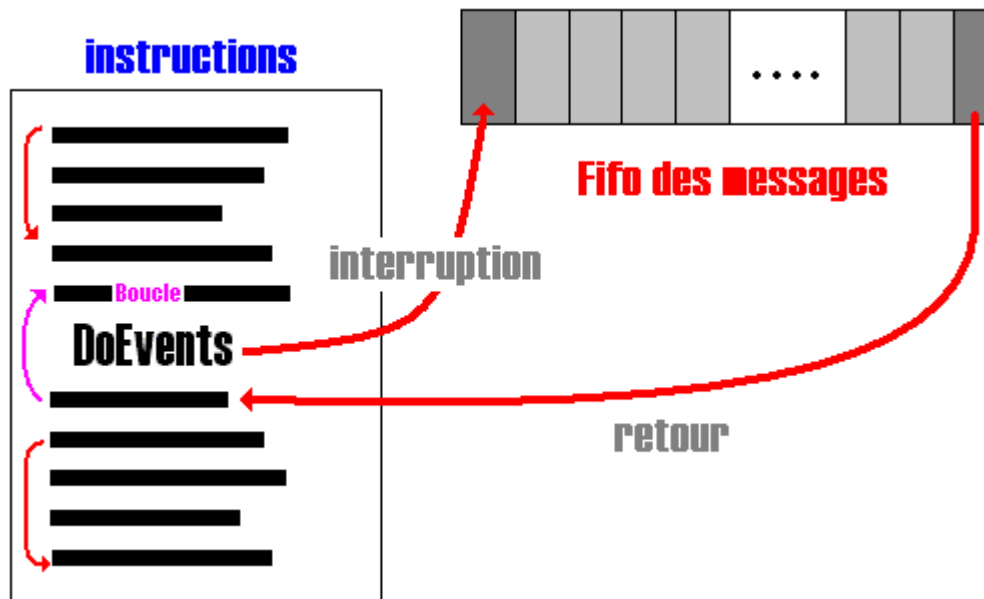
1.2.4 Que faire avec Application.DoEvents()

Nous allons utiliser la méthode de classe **DoEvents()** de la classe **Application** qui existe depuis Visual Basic 2, et qui permet de traiter tous les messages Windows présents dans la file d'attente des messages de la fenêtre (*elle passe la main au système d'une façon synchrone*) puis revient dans le programme qui l'a invoquée (identique à processMessages de Delphi).



Nous créons artificiellement une boucle en apparence infinie qui laisse le traitement des messages s'effectuer et qui attend qu'on lui demande de s'arrêter par l'intermédiaire d'un booléen **stop** dont la valeur change par effet de bord grâce à **DoEvents** :

```
static bool stop = false;
while (!stop) Application.DoEvents( );
```



Il suffit que lorsque **DoEvents()** s'exécute l'une des actions de traitement de messages provoque la mise du booléen **stop** à true pour que la boucle s'arrête et que le processus se termine.

Choisissons une telle action par exemple lorsque l'utilisateur clique sur le bouton de fermeture de la fiche, la fiche se ferme et l'événement **closed** est déclenché, **DoEvents()** revient dans la boucle d'attente **while (!stop) Application.DoEvents()** ; au tour de boucle suivant. Si lorsque l'événement **close** de la fiche a lieu nous en profitons pour mettre le booléen **stop** à true, dès le retour de **DoEvents()** le prochain tour de boucle arrêtera l'exécution de la boucle et le corps d'instruction de **Main** continuera à s'exécuter séquentiellement jusqu'à la fin (ici on arrêtera le processus).

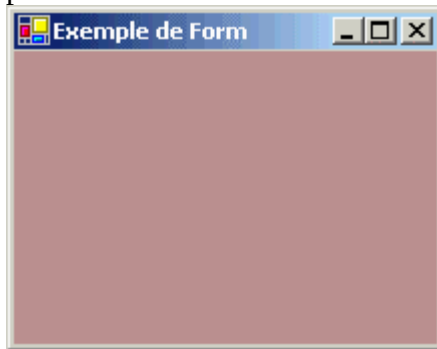
Ci-dessous le programme C# à mettre dans le fichier "**_exowin.cs**" :

```
using System;
using System.Windows.Forms;
namespace ApplicationTest
{
    public class Exemple {
        static bool stop = false; // le drapeau d'arrêt de la boucle d'attente

        static void fiche_Closed(object sender, System.EventArgs e) {
            // le gestionnaire de l'événement closed de la fiche
            stop = true;
        }

        static void Main() {
            System.Windows.Forms.Button button1 = new System.Windows.Forms.Button();
            Form fiche = new Form();
            fiche.Text="Exemple de Form";
            fiche.BackColor=System.Drawing.Color.RosyBrown;
            fiche.Closed += new System.EventHandler(fiche_Closed); // abonnement du gestionnaire
            fiche.Show();
            while (!stop) Application.DoEvents(); // boucle d'attente
            //... suite éventuelle du code avant arrêt
        }
    }
}
```

Lorsque nous compilons puis exécutons ce programme la fiche apparaît correctement et reste présente sur l'écran :



Elle se ferme et disparaît lorsque nous cliquons sur le bouton de fermeture.

On peut aussi vouloir toujours en utilisant la boucle infinie qui laisse le traitement des messages s'effectuer ne pas se servir d'un booléen et continuer après la boucle, mais plutôt essayer d'interrompre et de terminer l'application directement dans la boucle infinie sans exécuter la suite du code. La classe Application ne permet pas de terminer le processus.

Attention à l'utilisation de la méthode Exit de la classe Application qui semblerait être utilisable dans ce cas, en effet cette méthode arrête toutes les boucles de messages en cours sur tous les threads et ferme toutes les fenêtres de l'application; mais cette méthode ne force pas la fermeture de l'application.

Pour nous en convaincre compilons et exécutons le programme ci-après dans lequel l'événement `fiche_Closed` appelle `Application.Exit()`

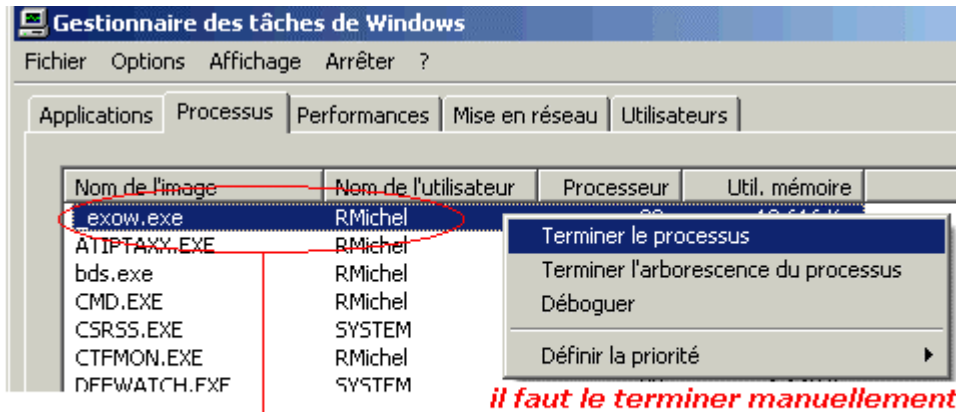
Ci-dessous le programme C# à mettre dans le fichier "`_exowin.cs`" :

```
using System;
using System.Windows.Forms;
namespace ApplicationTest
{
    public class Exemple
    {
        static void fiche_Closed (object sender, System.EventArgs e)
        { // le gestionnaire de l'événement closed de la fiche
            Application.Exit(); // on ferme la fenêtre, mais on ne termine pas le processus
        }

        static void Main() {
            System.Windows.Forms.Button button1 = new System.Windows.Forms.Button();
            Form fiche = new Form();
            fiche.Text="Exemple de Form";
            fiche.BackColor=System.Drawing.Color.RosyBrown;
            fiche.Closed += new System.EventHandler(fiche_Closed); // abonnement du gestionnaire
            fiche.Show();
            while (true) Application.DoEvents(); // boucle infinie
        }
    }
}
```

Lorsque nous compilons puis exécutons ce programme la fiche apparaît correctement et reste

présente sur l'écran, puis lorsque nous fermons la fenêtre comme précédemment, elle disparaît, toutefois le processus _exow.exe est toujours actif (la boucle tourne toujours, mais la fenêtre a été fermée) en faisant apparaître le gestionnaire des tâches de Windows à l'onglet processus nous voyons qu'il est toujours présent dans la liste des processus actifs. Si nous voulons l'arrêter il faut le faire manuellement comme indiqué ci-dessous :



Le processus est toujours en cours d'exécution

Comment faire pour réellement tout détruire ?

Il faut pouvoir détruire le processus en cours (en prenant soin d'avoir tout libéré avant si nécessaire), pour cela le NetFrameWork dispose d'une classe Process qui permet l'accès à des processus locaux ainsi que distants, et vous permet de démarrer et d'arrêter des processus systèmes locaux.

Nous pouvons connaître le processus en cours d'activation (ici, c'est notre application_exow.exe) grâce à la méthode de classe GetCurrentProcess et nous pouvons "tuer" un processus grâce à la méthode d'instance Kill :

```
static void fiche_Closed (object sender, System.EventArgs e)
{ // le gestionnaire de l'événement closed de la fiche
    System.Diagnostics.Process ProcCurr = System.Diagnostics.Process.GetCurrentProcess( );
    ProcCurr.Kill( );
}
```

Ci-dessous le programme C# à mettre dans le fichier "_exowin.cs" :

```
using System;
using System.Windows.Forms;
namespace ApplicationTest
{
    public class Exemple
    {
        static void fiche_Closed (object sender, System.EventArgs e)
        { // le gestionnaire de l'événement closed de la fiche
            System.Diagnostics.Process ProcCurr = System.Diagnostics.Process.GetCurrentProcess( );
            ProcCurr.Kill( );
        }

        static void Main( ) {
```

```

System.Windows.Forms.Button button1 = new System.Windows.Forms.Button( );
Form fiche = new Form( );
fiche.Text="Exemple de Form";
fiche.BackColor=System.Drawing.Color.RosyBrown;
fiche.Closed += new System.EventHandler(fiche_Closed); // abonnement du gestionnaire
fiche.Show( );
while (true) Application.DoEvents( ); // boucle infinie
    }
}

```

Après compilation, exécution et fermeture en faisant apparaître le gestionnaire des tâches de Windows à l'onglet processus nous voyons que le processus a disparu de la liste des processus actifs du système. Nous avons donc bien interrompu la boucle infinie.

Toutefois la console n'est pas l'outil préférentiel de C# dans le sens où C# est l'outil de développement de base de .Net et que cette architecture a vocation à travailler essentiellement avec des fenêtres.

Dans ce cas nous avons tout intérêt à utiliser un RAD visuel C# pour développer ce genre d'applications (comme l'on utilise Delphi pour le développement d'IHM en pascal objet). Une telle utilisation nous procure le confort du développement visuel, la génération automatique d'une bonne partie du code répétitif sur une IHM, l'utilisation et la réutilisation de composants logiciels distribués sur le net.

RAD utilisables

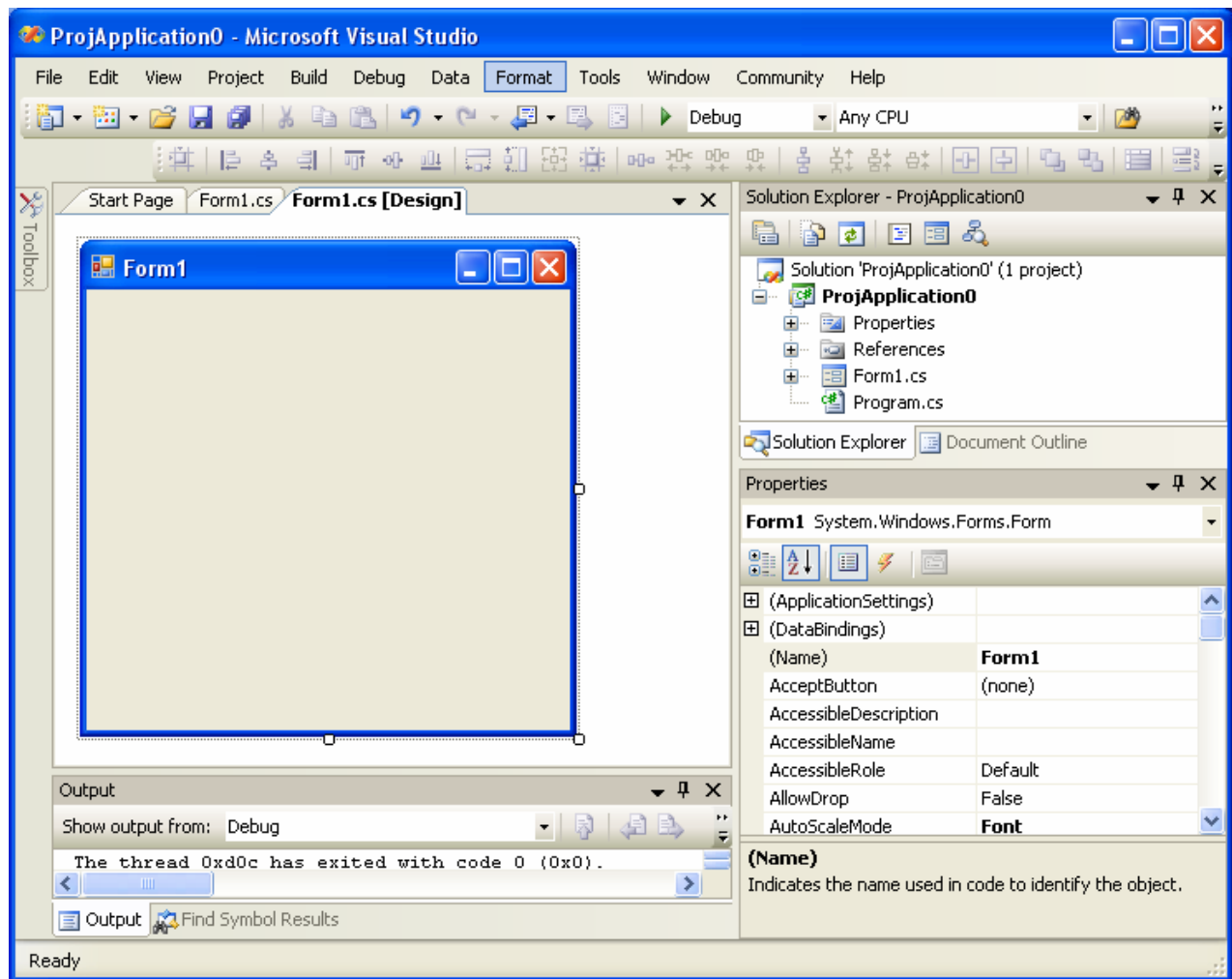
- **Visual Studio de microsoft** contient deux RAD de développement pour .Net, VBNet (fondé sur Visual Basic réellement objet et entièrement rénové) et Visual C# (fondé sur le langage C#), parfaitement adapté à .Net. (*versions express gratuites*)
- **C# Builder de Borland** reprenant les fonctionnalités de Visual C# inclus dans Borland Studio 2006 (*versions personnelle gratuite*)
- **Le monde de l'open source** construit un produit nommé **sharpDevelop** fournit à tous gratuitement aussi les mêmes fonctions que Visuel C# express de Microsoft, mais il est généralement en retard sur les versions de Microsoft.

1.3 un formulaire en C# est une fiche

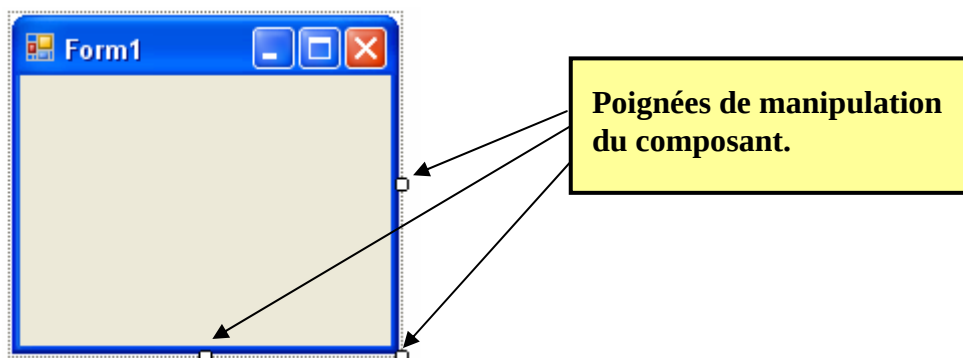
Les fiches ou formulaires C# représentent l'interface utilisateur (IHM) d'une application sous l'apparence d'une fenêtre visuelle. Comme les deux environnements RAD, Visual studio C# de Microsoft et C# Builder de Borland permettent de concevoir visuellement des applications avec IHM, nous dénommerons l'un ou l'autre par le terme général RAD C#.

Etant donné la disponibilité gratuite des trois RAD cités plus haut, dans cette partie de l'ouvrage nous avons utilisé C# Builder en version personnelle (très suffisante pour déjà écrire de bonnes applications) nous illustrerons donc tous nos exemples avec ce RAD. Il est évident que si le lecteur utilise un autre RAD, il ne sera pas dépaycé car les ergonomies de ces 3 RAD sont très proches.

Voici l'apparence d'un formulaire (ou fiche) dans le RAD Visuel C# en mode conception :



La fiche elle-même est figurée par l'image ci-dessous retaillable à volonté à partir de cliqué glissé sur l'une des huit petites "poignées carrées" situées aux points cardinaux de la fiche :



Ces formulaires sont en faits des objets d'une classe nommée **Form** de l'espace des noms **System.Windows.Forms**. Ci-dessous la hiérarchie d'héritage de Object à Form :

System.Object

System.MarshalByRefObject
System.ComponentModel.Component
 System.Windows.Forms.Control
 System.Windows.Forms.ScrollableControl
 System.Windows.Forms.ContainerControl
System.Windows.Forms.Form

La classe Form est la classe de base de tout style de fiche (ou formulaire) à utiliser dans votre application (statut identique à TForm dans Delphi) : fiche de dialogue, sans bordure etc..

Les différents styles de fiches sont spécifiés par l'énumération FormBorderStyle :

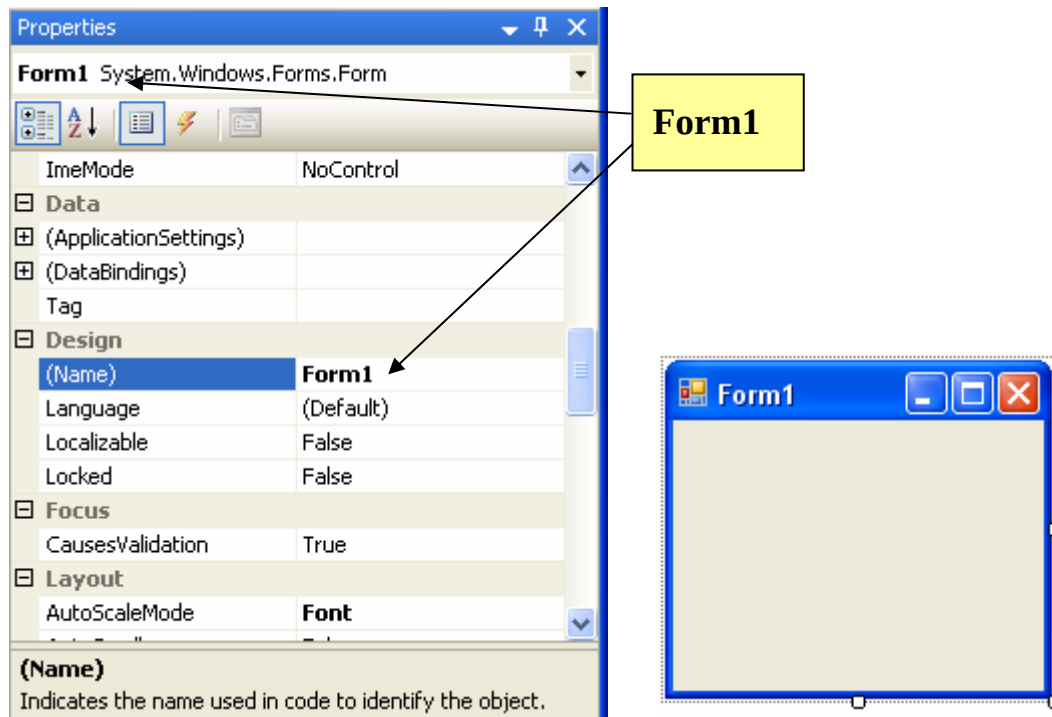
```
public Enum FormBorderStyle {Fixed3D, FixedDialog, FixedSingle, FixedToolWindow,
None, Sizable, SizableToolWindow }
```

Dans un formulaire, le style est spécifié par la **propriété FormBorderStyle** de la classe Form :

```
public FormBorderStyle FormBorderStyle {get; set;}
```

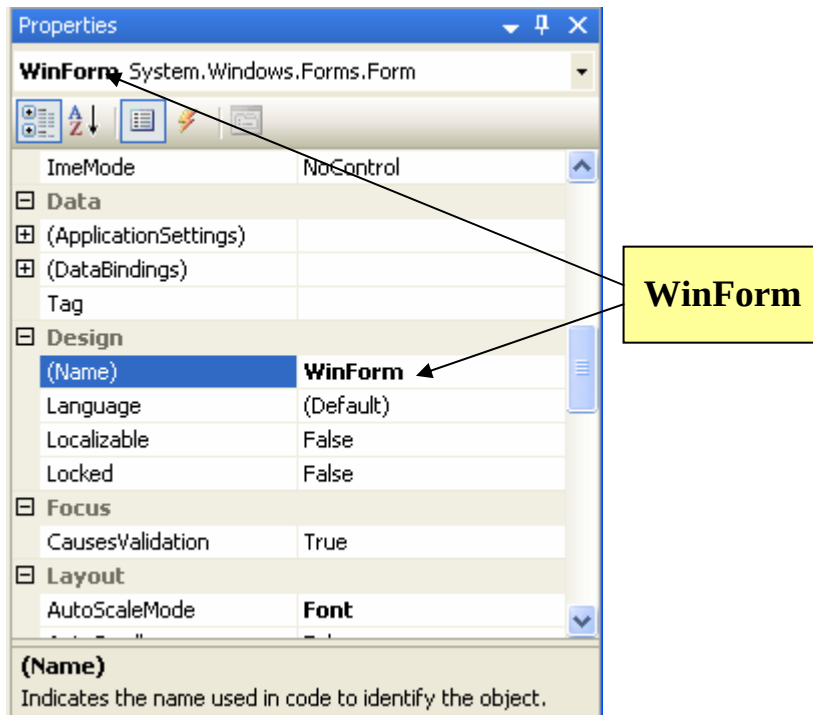
Toutes les propriétés en lecture et écriture d'une fiche sont accessibles à travers l'inspecteur d'objet qui répercute immédiatement en mode conception toute modification. Certaines provoquent des changements visuels d'autres non :

1°) Nous changeons le nom d'identificateur de notre fiche nommée **Form1** dans le programme en modifiant sa propriété Name dans l'inspecteur d'objet :

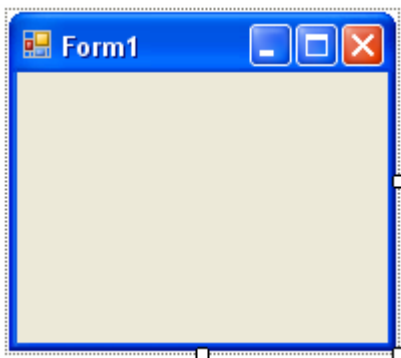


Le RAD construit automatiquement notre fiche principale comme une classe héritée de la classe Form et l'appelle Form1 : **public class WinForm : System.Windows.Forms.Form**{ ... }

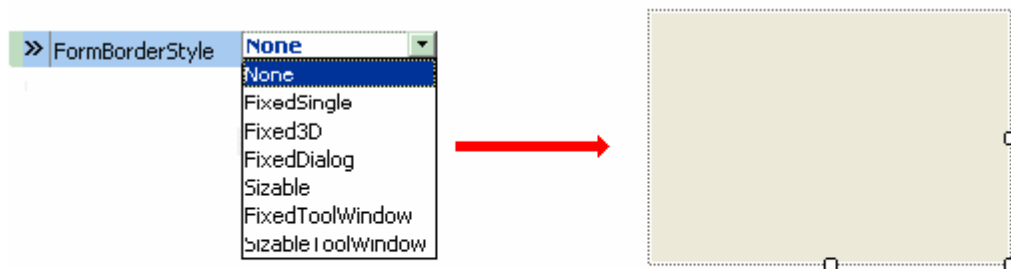
Après modification de la propriété Name par exemple par le texte **WinForm**, nous obtenons :



La classe de notre formulaire s'appelle désormais **WinForm**, mais son aspect visuel est resté le même :



2°) Par exemple sélectionnons dans l'inspecteur d'objet de Visuel C#, la propriété `FormBorderStyle` (le style par défaut est `FormBorderStyle.Sizable`) modifions la à la valeur **None** et regardons dans l'onglet conception la nouvelle forme de la fiche :



**la propriété change
de valeur**

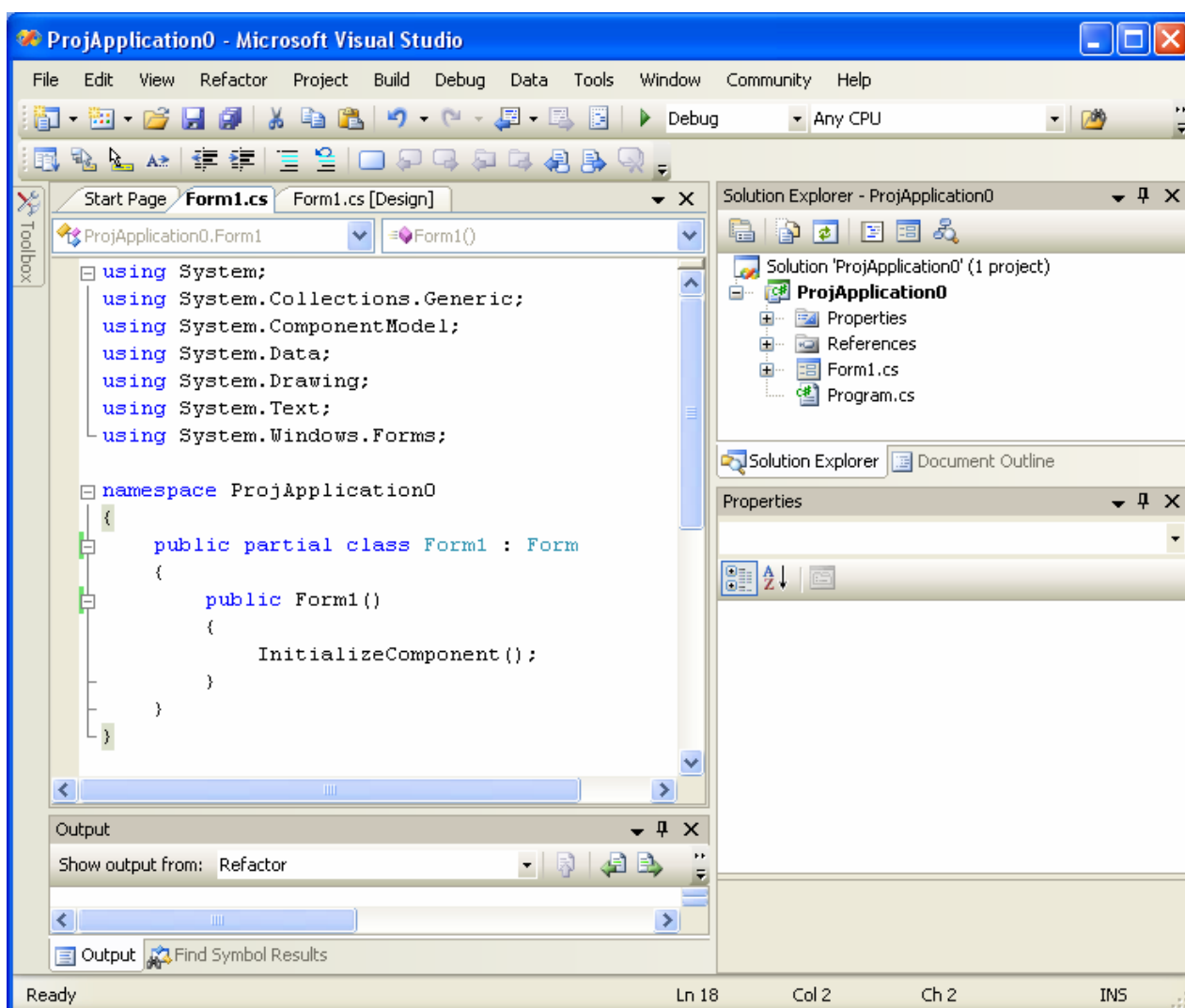


**le formulaire change
de forme**

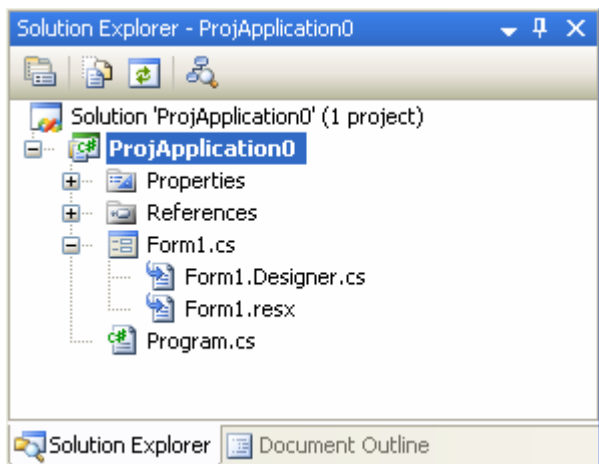
L'aspect visuel du formulaire a changé.

1.4 code C# engendré par le RAD pour un formulaire

Après avoir remis grâce à l'inspecteur d'objet, la propriété `FormBorderStyle` à sa valeur `Sizable` et remis le `Name` à la valeur initiale `Form1`, voyons maintenant en supposant avoir appelé notre application **ProjApplication0** ce que Visual C# a engendré comme code source que nous trouvons dans l'onglet de code `Form1.cs` pour notre formulaire :

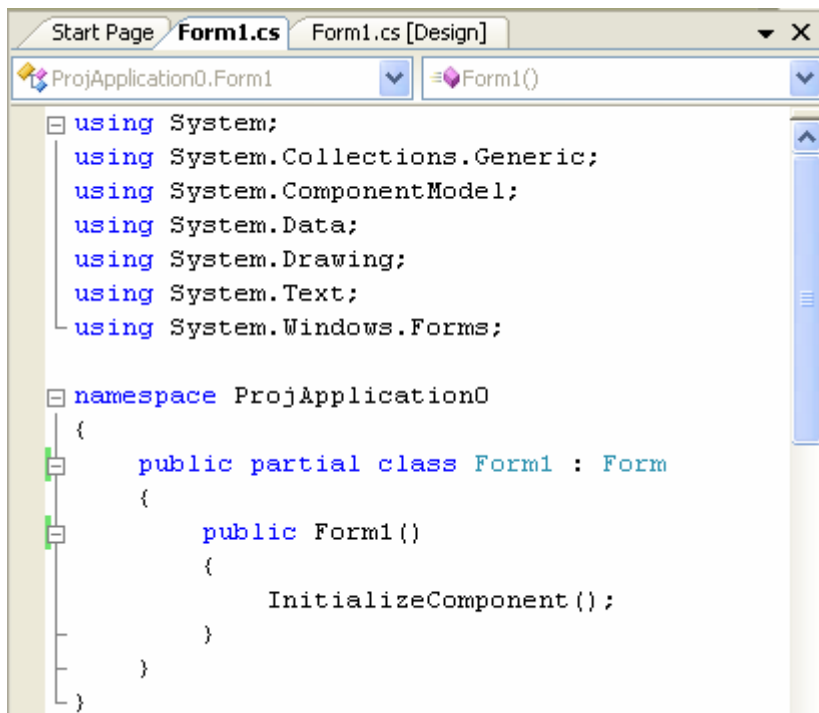


Le RAD Visuel C#, depuis la version 2.0, stocke les informations décrivant le programme dans 3 fichiers séparés (ici : Form1.cs, Form1.Designer.cs, Program.cs) :



La description de la **classe Form1** associée à la fiche est répartie dans les deux fichiers **Form1.cs** et **Form1.Designer.cs**.

Le fichier **Form1.cs** est celui qui contient les codes des gestionnaires d'événements que le développeur écrit au cours du codage :

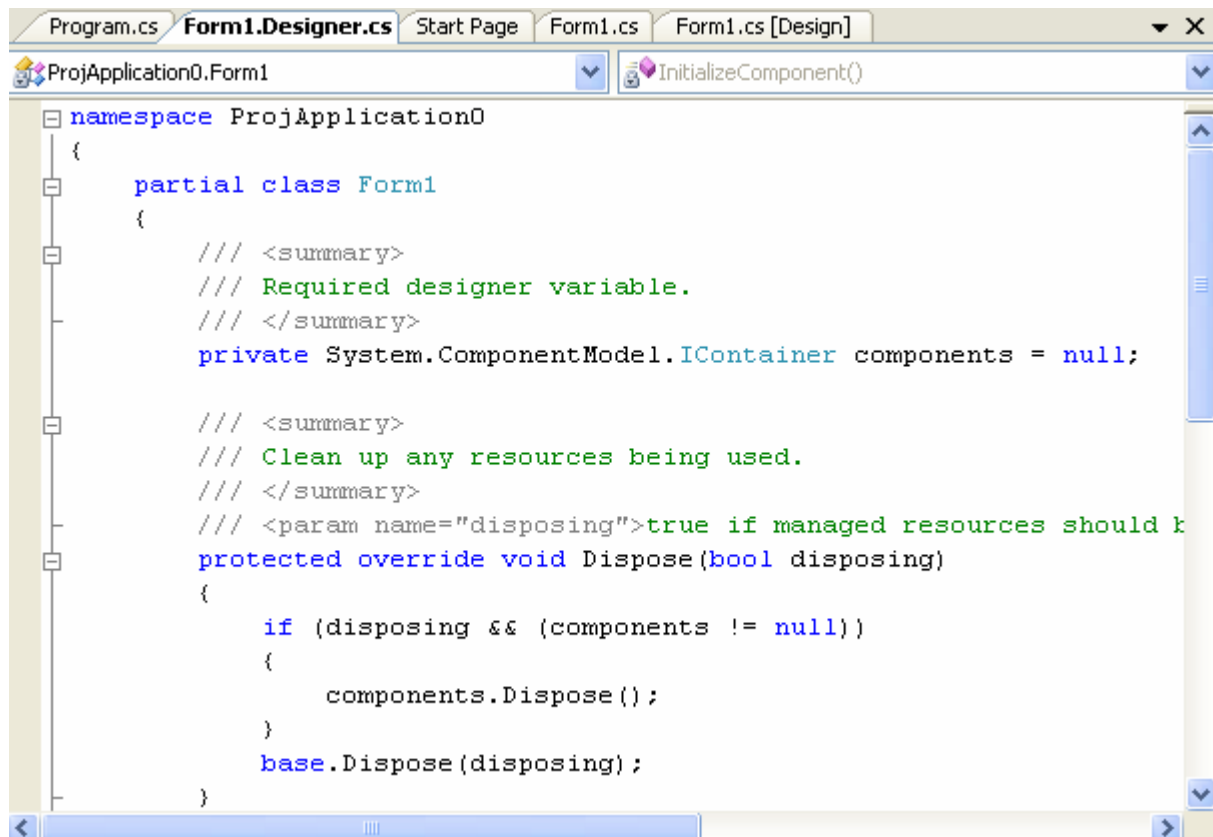


Au départ ce fichier ne contient que le constructeur de la classe Form1: public Form1().

Ce constructeur fait appel à la méthode "`private void InitializeComponent()`" dont le code est situé dans le fichier **Form1.Designer.cs** et qui a pour but de décrire les composants à visualiser avec la fiche Form1 et la fiche Form1 elle-même.

Le fichier **Form1.Designer.cs** est celui qui contient les codes générés automatiquement par Visual C# pour décrire, la fiche, les composants, les propriétés, les delegate pour lier aux composants les gestionnaires d'événements qui sont écrits par le développeur dans le fichier **Form1.cs**. Enfin, ce

fichier contient aussi le code servant à permettre le nettoyage des ressources utilisées, s'il elles existent.



En "recollant" artificiellement à titre d'illustration, les contenus respectifs des deux fichiers **Form1.Designer.cs** et **Form1.cs** décrivant la classe Form1, on obtient en fait la description complète de cette classe :

class Form1

{

```
public Form1()
{
    InitializeComponent();
}
```

Provient de Form1.cs

```
private System.ComponentModel.IContainer components = null;
protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();
    }
    base.Dispose(disposing);
}
private void InitializeComponent()
```

Provient de Form1.Designer.cs

```

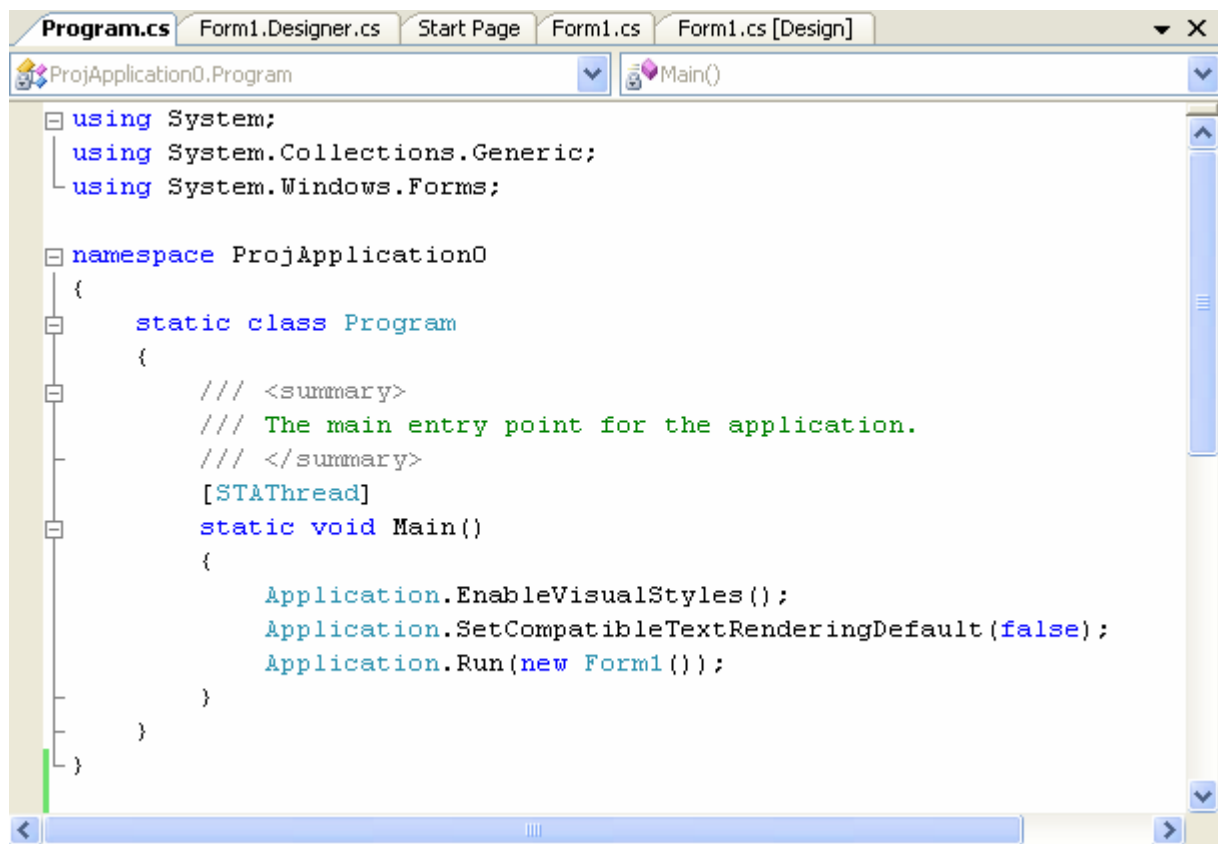
        this.SuspendLayout();
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(292, 266);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
    }

```

Provient de Form1.Designer.cs

// fin de la classe Form1

Le troisième fichier utilisé par le RAD se nomme **Program.cs**, il contient le code de la classe "principale" nommée Program, permettant le lancement de l'exécution du programme lui-même :



Quelques éléments explicatifs de l'analyse du code :

La méthode	correspond
<pre> static void Main() { Application.EnableVisualStyles(); Application.SetCompatibleTextRenderingDefault(false); Application.Run (new Form1 ()); } </pre>	Point d'entrée d'exécution de l'application, son corps contient un appel de la méthode statique Run de la classe Application, elle instancie un objet "new Form1 ()" de classe Form1 passé en paramètre à la méthode Run : c'est la fiche principale de l'application.

`Application.Run ( new Form1 ( ) );`

La classe `Application` (semblable à `TApplication` de Delphi) fournit des membres statiques (propriétés et méthodes de classes) pour gérer une application (démarrer, arrêter une application, traiter des messages Windows), ou d'obtenir des informations sur une application. Cette classe est **sealed** et ne peut donc pas être héritée.

La méthode `Run` de la classe `Application` dont voici la signature :
public static void `Run( ApplicationContext context );`

Exécute une boucle de messages d'application standard sur le thread en cours, par défaut le paramètre `context` écoute l'événement **Closed** sur la fiche principale de l'application et dès lors arrête l'application.

Pour les connaisseurs de Delphi , le démarrage de l'exécution s'effectue dans le programme principal :

```
program Project1;
uses Forms, Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
  Application.Initialize;
  Application.CreateForm (WinForm , Form1);
  Application.Run;
end.
```

Pour les connaisseurs des Awt et des Swing de Java, cette action C# correspond aux lignes suivantes :

Java2 avec Awt

```
class WinForm extends Frame {
  public WinForm ( ) {
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
  }
  protected void processWindowEvent(WindowEvent e) {
    super.processWindowEvent(e);
    if(e.getID() == WindowEvent.WINDOW_CLOSING) {
      System.exit(0); }
  }
  public static void main(String[] x )  {
    new WinForm ( );
  }
}
```

Java2 avec Swing

```
class WinForm extends JFrame {
  public WinForm ( ) {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  public static void main(String[] x )  {
    new WinForm ( );
  }
}
```

Lorsque l'on regarde de plus près le code de la classe **Form1** situé dans **Form1.Designer.cs** on se rend compte qu'il existe une ligne en grisé située après la méthode **Dispose** :

```
namespace ProjApplication0
{
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }
    }
}
```

Provient de Form1.Designer.cs

Windows Form Designer generated code

Il s'agit en fait de code replié (masqué) et généré automatiquement par le RAD, il est déconseillé de le modifier. Si nous le déplaçons et nous voyons apparaître la méthode privée `InitializeComponent()` contenant du code qui a manifestement été généré directement. En outre cette méthode est appelée dans le constructeur d'objet `Form1` :

Voici le contenu exact de l'onglet code avec sa zone de code replié :

```
#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.SuspendLayout();
    //
    // Form1
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(292, 266);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
}
#endregion
```

Lors du lancement du programme, la méthode `Main()` appelle le constructeur `Form1()` :

```
static void Main() {
    ...
    Application.Run ( new Form1 () );
}
```

le constructeur `Form1()` appelle la méthode `InitializeComponent()` :

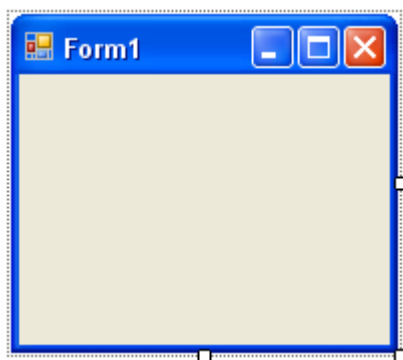
```

class Form1
{
    public Form1()
    {
        InitializeComponent();
    }
    private System.ComponentModel.IContainer components = null;
    protected override void Dispose(bool disposing)
    {
        if (disposing && (components != null))
        {
            components.Dispose();
        }
        base.Dispose(disposing);
    }

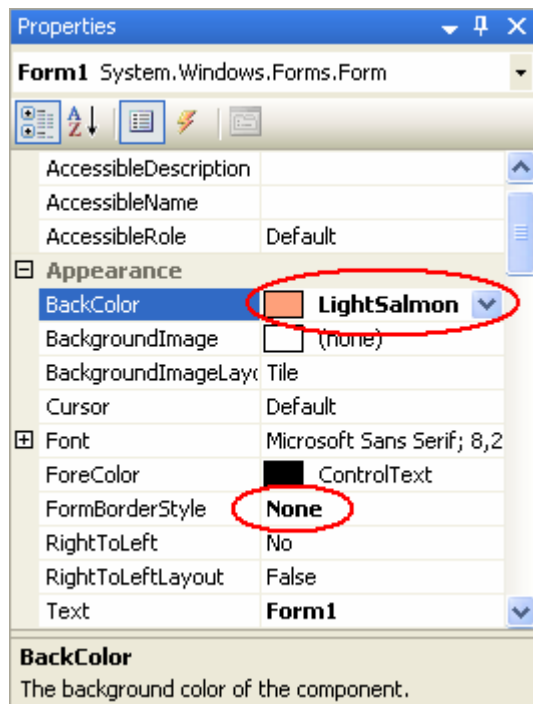
    private void InitializeComponent()
    {
        this.SuspendLayout();
        //
        // Form1
        //
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(292, 266);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);
    }
}

```

La fiche Form1 est alors affichée sur l'écran :



Essayons de voir comment une manipulation visuelle engendre des lignes de code, pour cela modifions dans l'inspecteur d'objet deux propriétés **FormBorderStyle** et **BackColor**, la première est mise à **None** la seconde qui indique la couleur du fond de la fiche est mise à **LightSalmon** :



Consultons après cette opération le contenu du nouveau code généré, nous trouvons deux nouvelles lignes de code correspondant aux nouvelles actions visuelles effectuées (**les nouvelles lignes sont figurées en rouge**) :

```
#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.SuspendLayout();
    //
    // Form1
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.BackColor = System.Drawing.Color.LightSalmon;
    this.ClientSize = new System.Drawing.Size(292, 266);
    this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.None;
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
}
#endregion
```

1.5 Libération de ressources non managées

Dans le code engendré par Visual studio ou C# Builder, nous avons laissé de côté la méthode Dispose :

```
protected override void Dispose (bool disposing)
{
    if (disposing && (components != null) ) {
        components.Dispose( );
    }
    base.Dispose( disposing );
}
```

Pour comprendre son utilité, il nous faut avoir quelques lumières sur la façon que NetFrameWork a de gérer les ressources, rappelons que le CLR exécute et gère le code administré c'est à dire qu'il vérifie la validité de chaque action avant de l'exécuter. Le code non administré ou **ressource non managée** en C# est essentiellement du code sur les pointeurs qui doivent être déclarés **unsafe** pour pouvoir être utilisés, ou bien du code sur des fichiers, des flux, des handles.

La méthode **Dispose** existe déjà dans la classe mère **System.ComponentModel.Component** sous forme de deux surcharges avec deux signatures différentes. Elle peut être utile si vous avez mobilisé des ressources personnelles (on dit aussi **ressources non managées**) et que vous souhaitiez que celles-ci soient libérées lors de la fermeture de la fiche :

classe : **System.ComponentModel.Component**

méthode : **public virtual void** Dispose();

Libère **toutes** les ressources utilisées par Component.

méthode : **protected virtual void** Dispose(bool disposing);

Libère **uniquement** les ressources **non managées** utilisées par Component..

ou

Libère **toutes** les ressources utilisées par Component.

Selon la valeur de disposing

- disposing = **true** pour libérer **toutes** les ressources (managées et non managées) ;
- disposing = **false** pour libérer **uniquement** les ressources **non managées**.

Remarque-1

Notons que pour le débutant cette méthode ne sera jamais utilisée et peut être omise puisqu'il s'agit d'une surcharge dynamique de la méthode de la classe mère.

Remarque-2

Il est recommandé par Microsoft, qu'un objet Component libère des ressources explicitement en appelant sa méthode Dispose sans attendre une gestion automatique de la mémoire lors d'un appel implicite au Garbage Collector.

Si nous voulons comprendre comment fonctionne le code engendré pour la méthode **Dispose**, il nous faut revenir à des éléments de base de la gestion mémoire en particulier relativement à la libération des ressources par le ramasse-miettes (garbage collector).

1.6 Comment la libération a-t-elle lieu dans le NetFramework ?

La classe mère de la hiérarchie dans le **NetFramework** est la classe **System.Object**, elle possède une méthode virtuelle **protected** **Finalize**, qui permet de libérer des ressources et d'exécuter d'autres opérations de nettoyage avant que **Object** soit récupéré par le garbage collecteur GC.

 *Bibliothèque de classes .NET Framework*
Object, membres

Méthodes protégées

 Finalize Pris en charge par le .NET Compact Framework.	Substitué. Autorise Object à tenter de libérer des ressources et d'exécuter d'autres opérations de nettoyage avant que Object soit récupéré par l'opération garbage collection. En C# et C++, les finaliseurs sont exprimés à l'aide de la syntaxe des destructeurs.
--	---

Lorsqu'un objet devient inaccessible il est automatiquement placé dans la **file d'attente de finalisation** de type FIFO, le garbage collecteur GC, lorsque la mémoire devient trop basse, effectue son travail en parcourant cette file d'attente de finalisation et en libérant la mémoire occupée par les objets de la file par appel à la méthode **Finalize** de chaque objet.

Donc si l'on souhaite libérer des ressources personnalisées, il suffit de redéfinir dans une classe fille la méthode **Finalize()** et de programmer dans le corps de la méthode la libération de ces ressources.

En C# on pourrait écrire pour une classe **MaClasse** :

```
protected override void Finalize() {  
    try {  
        // libération des ressources personnelles  
    }  
    finally  
    {  
        base.Finalize(); // libération des ressources du parent  
    }  
}
```

Mais syntaxiquement en C# la méthode **Finalize** n'existe pas et le code précédent, s'il représente bien ce qu'il faut faire, ne sera pas accepté par le compilateur. En C# la méthode **Finalize** s'écrit comme un destructeur de la classe **MaClasse** :

```
~MaClasse() {  
    // libération des ressources personnelles  
}
```

1.7 Peut-on influencer sur cette la libération dans le NetFramework ?

Le processus de gestion de la libération mémoire et de sa récupération est entièrement automatisé dans le CLR, mais selon les nécessités on peut avoir le besoin de gérer cette désallocation : il existe pour cela, une classe **System.GC** qui autorise le développeur à une certaine dose de contrôle du garbage collector.

Par exemple, vous pouvez empêcher explicitement la méthode Finalize d'un objet figurant dans la file d'attente de finalisation d'être appelée, (utilisation de la méthode : **public static void SuppressFinalize(object obj);**)

Bibliothèque de classes .NET Framework	
GC, membres	
Méthodes publiques	
 ReRegisterForFinalize Pris en charge par le .NET Compact Framework.	Demande à ce que le système appelle la méthode du finaliseur pour l'objet spécifié pour lequel SuppressFinalize a été précédemment appelé.
 SuppressFinalize Pris en charge par le .NET Compact Framework.	Demande à ce que le système n'appelle pas la méthode du finaliseur pour l'objet spécifié.

Vous pouvez aussi obliger explicitement la méthode Finalize d'un objet figurant dans la file d'attente de finalisation mais contenant GC.SuppressFinalize(...) d'être appelée, (utilisation de la méthode : **public static void ReRegisterForFinalize(object obj);**).

Microsoft propose deux recommandations pour la libération des ressources :

Il est recommandé d'empêcher les utilisateurs de votre application d'appeler directement la méthode Finalize d'un objet en limitant sa portée à protected.
Il est vivement déconseillé d'appeler une méthode Finalize pour une autre classe que votre classe de base directement à partir du code de votre application. Pour supprimer correctement des ressources non managées, il est recommandé d'implémenter une méthode Dispose ou Close publique qui exécute le code de nettoyage nécessaire pour l'objet.

Microsoft propose des conseils pour écrire la méthode Dispose :

1- La méthode Dispose d'un type doit libérer toutes les ressources qu'il possède.
2- Elle doit également libérer toutes les ressources détenues par ses types de base en appelant la méthode Dispose de son type parent. La méthode Dispose du type parent doit libérer toutes les ressources qu'il possède et appeler à son tour la méthode Dispose de son type parent, propageant ainsi ce modèle dans la hiérarchie des

types de base.
3- Pour que les ressources soient toujours assurées d'être correctement nettoyées, une méthode Dispose doit pouvoir être appelée en toute sécurité à plusieurs reprises sans lever d'exception.
4- Une méthode Dispose doit appeler la méthode GC.SuppressFinalize de l'objet qu'elle supprime.
5- La méthode Dispose doit être à liaison statique

1.8 Design Pattern de libération des ressources non managées

Le NetFrameWork propose une interface IDisposable ne contenant qu'une seule méthode : Dispose

 Bibliothèque de classes .NET Framework

IDisposable, membres

[IDisposable, vue d'ensemble](#)

Méthodes publiques

 Dispose Pris en charge par le .NET Compact Framework.	Exécute les tâches définies par l'application associées à la libération ou à la redéfinition des ressources non managées.
--	---

Rappel

Il est recommandé par Microsoft, qu'un objet Component libère des ressources explicitement en appelant sa méthode Dispose sans attendre une gestion automatique de la mémoire lors d'un appel implicite au Garbage Collector. C'est ainsi que fonctionnent tous les contrôles et les composants de NetFrameWork. Il est bon de suivre ce conseil car dans le modèle de conception fourni ci-après, la libération d'un composant fait libérer en cascade tous les éléments de la hiérarchie sans les mettre en liste de finalisation ce qui serait une perte de mémoire et de temps pour le GC.

Design Pattern de libération dans la classe de base

Voici pour information, proposé par Microsoft, un modèle de conception (Design Pattern) d'une classe MaClasseMere implémentant la mise à disposition du mécanisme de libération des ressources identique au NetFrameWork :

```
public class MaClasseMere : IDisposable {
    private bool disposed = false;
    // un exemple de ressource managée : un composant
    private Component Components = new Component( );
    // ...éventuellement des ressources non managées (pointeurs...)

    public void Dispose( ) {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    protected virtual void Dispose(bool disposing) {
        if (!this.disposed) {
            if (disposing) {
                Components.Dispose( ); // libère le composant
                // libère les autres éventuelles ressources managées
            }
            // libère les ressources non managées :
            //... votre code de libération
            disposed = true;
        }
        ~MaClasseMere ( ) { // finaliseur par défaut
            Dispose(false);
        }
    }
}
```

Ce modèle n'est présenté que pour mémoire afin de bien comprendre le modèle pour une classe fille qui suit et qui correspond au code généré par le RAD C# .

Design Pattern de libération dans une classe fille

Voici proposé le modèle de conception simplifié (Design Pattern) d'une classe `MaClasseFille` descendante de `MaClasseMere`, la classe fille contient une ressource de type `System.ComponentModel.Container`

```
public class MaClasseFille : MaClasseMere {
    private System.ComponentModel.Container components = null ;
    public MaClasseFille () {
        // code du constructeur....
    }

    protected override void Dispose(bool disposing) {
        if (disposing) {
            if (components != null) { // s'il y a réellement une ressource
                components.Dispose(); // on Dispose cette ressource
            }
            // libération éventuelle d'autres ressources managées...
        }
        // libération des ressources personnelles (non managées)...
        base.Dispose(disposing); // on Dispose dans la classe parent
    }
}
```

Information NetFrameWork sur la classe `Container` :

System.ComponentModel.Container

La classe **Container** est l'implémentation par défaut pour l'interface `IContainer`, une instance s'appelle un conteneur.

Les conteneurs sont des objets qui encapsulent et effectuent le suivi de zéro ou plusieurs composants qui sont des objets visuels ou non de la classe **System.ComponentModel.Component**.

Les références des composants d'un conteneur sont rangées dans une file FIFO, qui définit également leur ordre dans le conteneur.

La classe **Container** suit le modèle de conception mentionné plus haut quant à la libération des ressources managées ou non. Elle possède deux surcharges de `Dispose` implémentées selon le Design Pattern : la méthode `protected virtual void Dispose( bool disposing);` et la méthode `public void Dispose( )`. Cette méthode libère toutes les ressources détenues par les objets managés stockés dans la FIFO du `Container`. Cette méthode appelle la méthode `Dispose( )` de chaque objet référencé dans la FIFO.

Les formulaires implémentent **IDisposable** :

La classe **System.Windows.Forms.Form** hérite de la classe **System.ComponentModel.Component**, or si nous consultons la documentation technique nous constatons que la classe `Component` en autres spécifications implémente l'interface `IDisposable` :

```
public class Component : MarshalByRefObject, IComponent, IDisposable
```

Donc tous les composants de C# sont construits selon le Design Pattern de libération :

La classe Component implémente le Design Pattern de libération de la classe de base et toutes les classe descendantes dont la classe Form implémente le Design Pattern de libération de la classe fille.

Nous savons maintenant à quoi sert la méthode Dispose dans le code engendré par le RAD, elle nous propose une libération automatique des ressources de la liste des composants que nous aurions éventuellement créés :

```
// Nettoyage des ressources utilisées :
protected override void Dispose (bool disposing)
{
    if (disposing && (components != null) ) { // Nettoyage des ressources managées
        components.Dispose( );
    }
    // Nettoyage des ressources non managées
    base.Dispose(disposing);
}
```

1.9 Un exemple utilisant la méthode Dispose d'un formulaire

Supposons que nous avons construit un composant personnel dans une classe **UnComposant** qui hérite de la classe Component selon le Design Pattern précédent, et que nous avons défini cette classe dans le namespace ProjApplication0 :

```
System.ComponentModel.Component
    |__ProjApplication0.UnComposant
```

Nous voulons construire une application qui est un formulaire et nous voulons créer lors de l'initialisation de la fiche un objet de classe **UnComposant** que notre formulaire utilisera.

A un moment donné notre application ne va plus se servir du tout de notre composant, si nous souhaitons gérer la libération de la mémoire allouée à ce composant, nous pouvons :

- Soit attendre qu'il soit éligible au GC, en ce cas la mémoire sera libérée lorsque le GC le décidera,
- Soit le recenser auprès du conteneur de composants **components** (l'ajouter dans la FIFO de **components** si le conteneur a déjà été créé). Sinon nous créons ce conteneur et nous utilisons la méthode d'ajout (Add) de la classe **System.ComponentModel.Container** pour ajouter notre objet de classe **UnComposant** dans la FIFO de l'objet conteneur **components**.

```

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace ProjApplication0
{
    /// <summary>
    /// Description Résumé de Form1.
    /// </summary>
    public partial class Form1 : System.Windows.Forms.Form
    {
        /// <summary>
        /// Variable requise par le concepteur.
        /// </summary>
        private System.ComponentModel.Container components = null;
        private UnComposant MonComposant = new UnComposant( );

        public Form1 ( )
        {
            InitializeComponent( );

            if ( components == null )
            {
                components = new Container( );
                components.Add ( MonComposant );
            }
        }
        /// <summary>
        /// Nettoyage des ressources utilisées.
        /// </summary>
        protected override void Dispose (bool disposing)
        {
            if (disposing && (components != null) ) {
                components.Dispose( );
                //-- libération ici d'autres ressources managées...
            }
            //-- libération ici de vos ressources non managées...
            base.Dispose(disposing);
        }
    }
}

```

Déclaration-instanciation
d'un composant personnel.

Ajout du composant personnel
dans la Fifo de composants.

Notre composant personnel est
libéré avec les autres

region Code généré par le concepteur Windows Form

```

public class P`rogram
{
    /// <summary>
    /// Le point d'entrée principal de l'application.
    /// </summary>
    [STAThread]
    static void Main( )
    {
        ....
        Application.Run(new Form1 ( ));
    }
}

```

1.10 L'instruction **using** appelle **Dispose()**

La documentation technique signale que deux utilisations principales du mot clé **using** sont possibles :

Directive **using** :

Crée un alias pour un espace de noms ou importe des types définis dans d'autres espaces de noms.

*Ex: **using** System.IO ; **using** System.Windows.Forms ; ...*

Instruction **using** :

Définit une portée au bout de laquelle un objet est supprimé.

C'est cette deuxième utilisation qui nous intéresse : l'instruction **using**

```
<instruction using> ::= using ( <identif. Objet> | <liste de Déclar & instanciation> ) <bloc instruction>
<bloc instruction> ::= { < suite d'instructions > }
<identif. Objet> ::= un identificateur d'un objet existant et instancié
<liste de Déclar & instanciation> ::= une liste séparée par des virgules de déclaration et initialisation d'objets semblable à la partie initialisation d'une boucle for.
```

Ce qui nous donne deux cas d'écriture de l'instruction **using** :

1° - sur un objet déjà instancié :

```
classeA Obj = new classeA( );
....
using ( Obj )
{
    // code quelconque....
}
```

2° - sur un (des) objet(s) instancié(s) localement au **using** :

```
using ( classeB Obj1 = new classeB ( ), Obj2 = new classeB ( ), Obj3 = new classeB ( ) )
{
    // code quelconque....
}
```

Le **using** lance la méthode **Dispose** :

Dans les deux cas, on utilise une instance (Obj de classeA) ou l'on crée des instances (Obj1, Obj2 et Obj3 de classeB) dans l'instruction **using** pour garantir que la méthode **Dispose** est appelée sur l'objet lorsque l'instruction **using** est quittée.

Les objets que l'on utilise ou que l'on crée doivent implémenter l'interface **System.IDisposable**. Dans les exemples précédents classeA et classeB doivent implémenter elles-même ou par héritage l'interface **System.IDisposable**.

Exemple

Soit un objet visuel **button1** de classe **System.Windows.Forms.Button**, c'est la classe mère **Control** de **Button** qui implémente l'interface **System.IDisposable** :

```
public class Control : IComponent, IDisposable, IParserAccessor, IDataBindingsAccessor
```

Soient les lignes de code suivantes où **this** est une fiche :

```
// ....  
this.button1 = new System.Windows.Forms.Button ();  
using( button1) {  
    // code quelconque....  
}  
// suite du code ....
```

A la sortie de l'instruction **using** juste avant la poursuite de l'exécution de la suite du code, **button1.Dispose()** a été automatiquement appelée par le CLR (le contrôle a été détruit et les ressources utilisées ont été libérées immédiatement).

1.11 L'attribut [STAThread]

Nous terminons notre examen du code généré automatiquement par le RAD pour une application fenêtrée de base, en indiquant la signification de l'attribut (mot entre crochets **[STAThread]**) situé avant la méthode **Main** :

```
/// <summary>  
/// Le point d'entrée principal de l'application.  
/// </summary>  
[STAThread]  
static void Main()  
{  
    .....  
    Application.Run(new Form1 ( ));  
}
```

Cet attribut placé ici devant la méthode **Main** qualifie la manière dont le CLR exécutera l'application, il signifie : **Single Thread Apartments**.

Il s'agit d'un modèle de gestion mémoire où l'application et tous ses composants est gérée dans **un seul thread** ce qui évite des conflits de ressources avec d'autres threads. Le développeur n'a pas à s'assurer de la bonne gestion des éventuels conflits de ressources entre l'application et ses composants.

Si on omet cet attribut devant la méthode Main, le CLR choisi automatiquement **[MTAThread]** **Multi Thread Apartments**, modèle de mémoire dans lequel l'application et ses composants sont gérés par le CLR en plusieurs thread, le développeur doit alors s'assurer de la bonne gestion des éventuels conflits de ressources entre l'application et ses composants.

Sauf nécessité d'augmentation de la fluidité du code, il faut laisser (ou mettre en mode console) l'attribut **[STAThread]** :

```
...  
[STAThread]  
static void Main( )  
{  
    Application.Run(new Form1 ( ));  
}
```

Des contrôles dans les formulaires



Plan général:

1. Les contrôles et les fonds graphiques

- 1.1 Les composants en général
- 1.2 Les contrôles sur un formulaire
- 1.3 Influence de la propriété parent sur l'affichage visuel d'un contrôle
- 1.4 Des graphiques dans les formulaires avec le GDI+
- 1.5 Le dessin doit être persistant
- 1.6 Deux exemples de graphiques sur plusieurs contrôles
 - méthode générale pour tout redessiner
 - des dessins sur événements spécifiques

1. Les contrôles et les fonds graphiques

Nous renvoyons le lecteur à la documentation du constructeur pour la manipulation de l'interface du RAD qu'il aura choisi. Nous supposons que le lecteur a acquis une dextérité minimale dans cette manipulation visuelle (déposer des composants, utiliser l'inspecteur d'objet (ou inspecteur de propriétés), modifier des propriétés, créer des événements. Dans la suite du document, nous portons notre attention sur le code des programmes et sur leur comportement.

Car il est essentiel que le **lecteur sache par lui-même écrire le code** qui sera généré automatiquement par un RAD, sous peine d'être prisonnier du RAD et de ne pas pouvoir intervenir sur ce code !

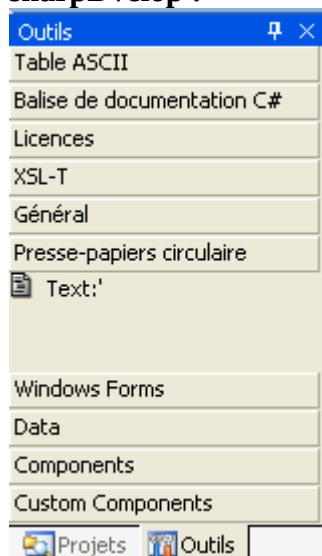
1.1 les composants en général

System.Windows.Forms.Control définit la classe de base des contrôles qui sont des composants avec représentation visuelle, les fiches en sont un exemple.

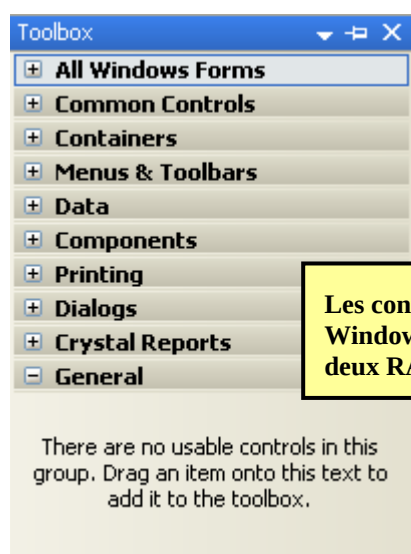
```
System.Object
System.MarshalByRefObject
System.ComponentModel.Component
System.Windows.Forms.Control
System.Windows.Forms.ScrollableControl
System.Windows.Forms.ContainerControl
System.Windows.Forms.Form
```

Dans les RAD, la programmation visuelle des contrôles a lieu d'une façon très classique, par glisser déposer de composants situés dans une palette ou boîte d'outils. Il est possible de déposer visuellement des composants, certains sont visuels ils s'appellent contrôles, d'autres sont non visuels, ils s'appellent seulement composants.

sharpDevelop :

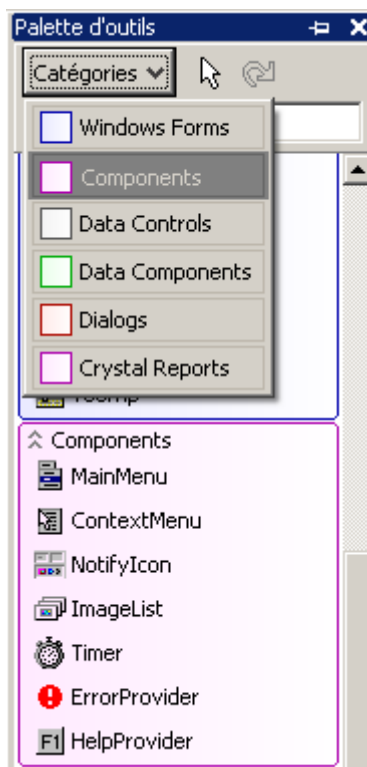


Visual C# :

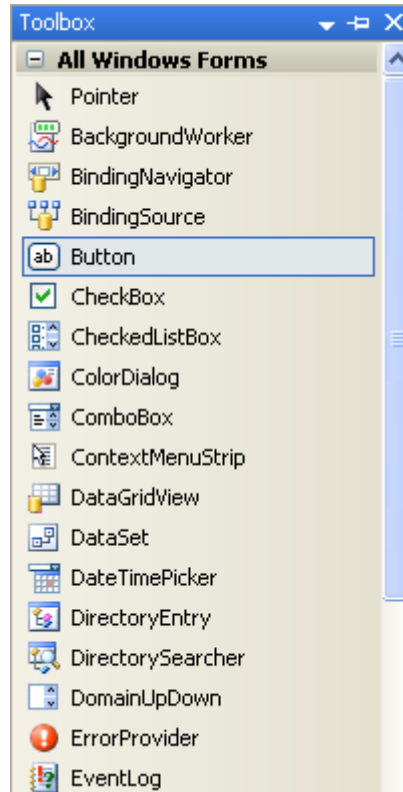


Les contrôles visuels ou Windows Forms dans deux RAD.

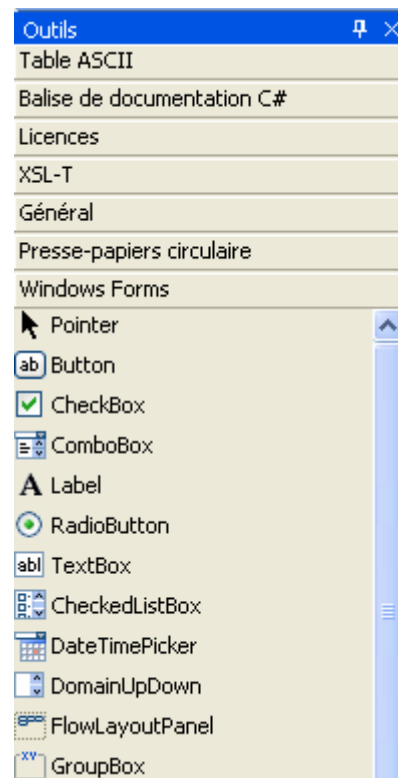
C# Builder



Visual C# :



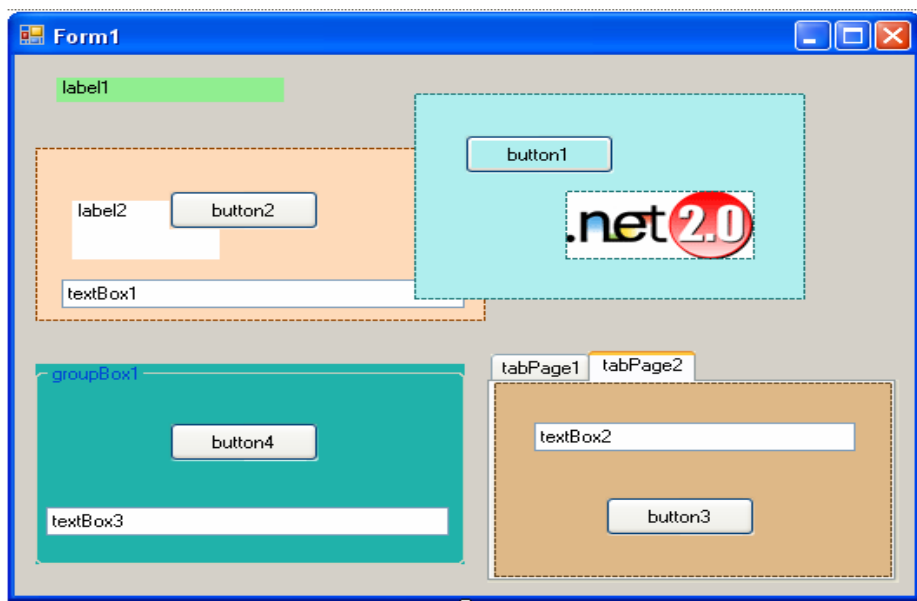
sharpdevelop :



Pour pouvoir construire une IHM, il nous faut pouvoir utiliser à minima les composants visuels habituels que nous retrouvons dans les logiciels windows-like. Ici aussi la documentation technique fournie avec le RAD détaillera les différentes entités mises à disposition de l'utilisateur.

1.2 les contrôles sur un formulaire

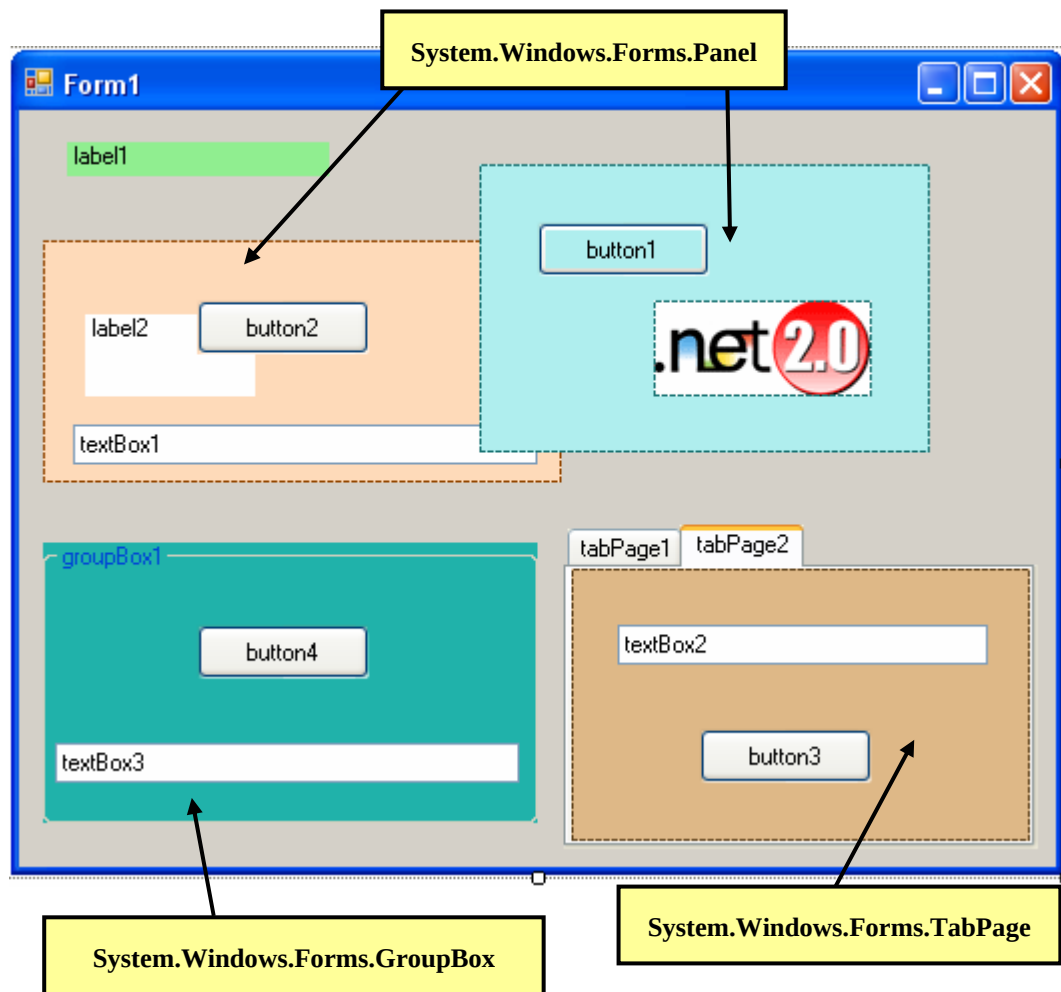
Voici un exemple de fiche comportant 7 catégories de contrôles différents :



Il existe des contrôles qui sont des conteneurs visuels, les quatre classes ci-après sont les principales classe de conteneurs visuels de C# :

```
System.Windows.Forms.Form  
System.Windows.Forms.Panel  
System.Windows.Forms.GroupBox  
System.Windows.Forms.TabPage
```

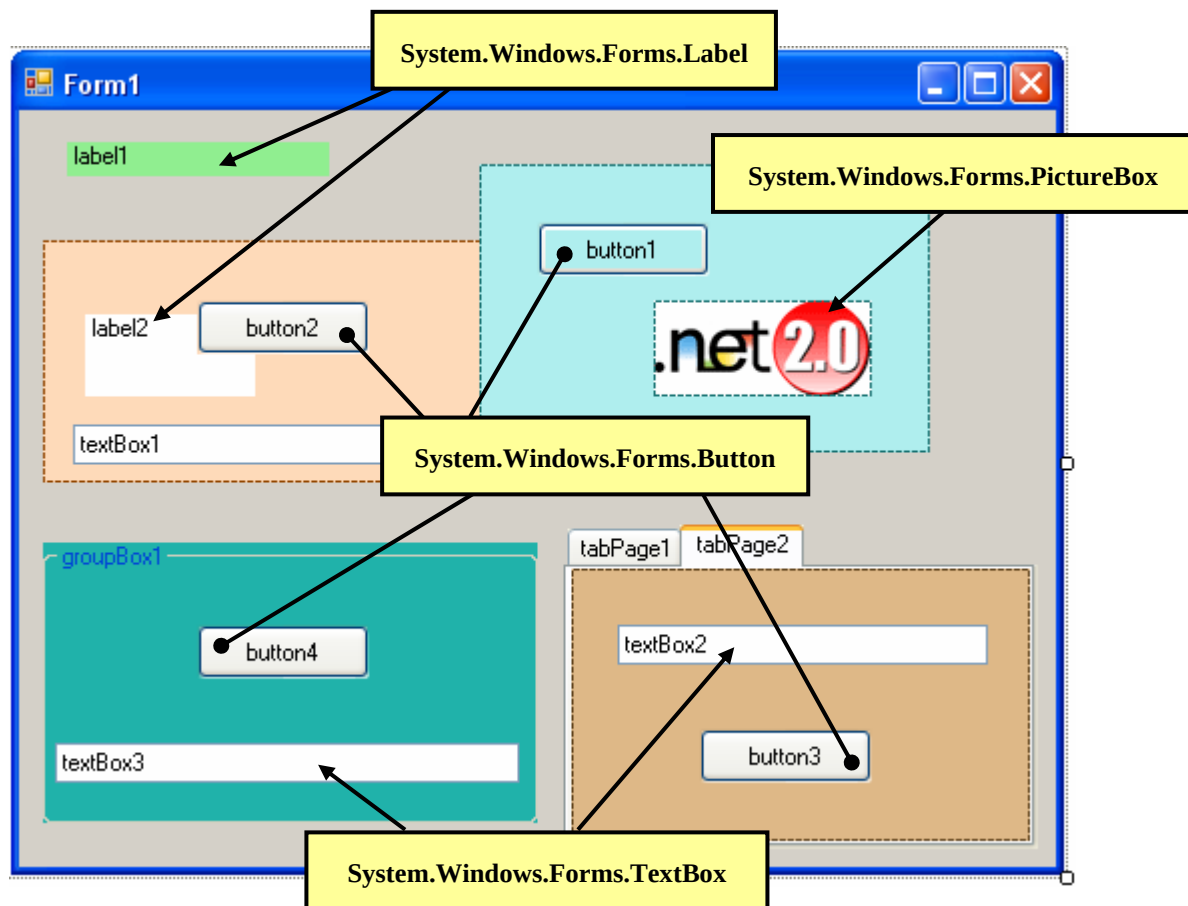
Sur la fiche précédente nous relevons outre le formulaire lui-même, quatre conteneurs visuels répartis en trois catégories de conteneurs visuels :



Un conteneur visuel permet à d'autres contrôles de s'afficher sur lui et lui communique par lien de parenté des valeurs de propriétés par défaut (police de caractères, couleur du fond,...). Un objet de chacune de ces classes de conteneurs visuels peut être le "parent" de n'importe quel contrôle grâce à sa propriété `Parent` qui est en lecture et écriture :

```
public Control Parent {get; set;}  
C'est un objet de classe Control qui représente le conteneur visuel du contrôle.
```

Sur chaque conteneur visuel a été déposé un contrôle de classe **System.Windows.Forms.Button** qui a "hérité" par défaut des caractéristiques de police et de couleur de fond de son parent. Ci-dessous les classes de tous les contrôles déposés :



Le code C# engendré dans "**Form1.Designer.cs**" pour cette interface :

```
namespace ProjApplication0
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
        }
    }
}
```

```
base.Dispose(disposing);
}
```

#region Windows Form Designer generated code

```
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.label1 = new System.Windows.Forms.Label();
    this.panel1 = new System.Windows.Forms.Panel();
    this.button2 = new System.Windows.Forms.Button();
    this.label2 = new System.Windows.Forms.Label();
    this.button1 = new System.Windows.Forms.Button();
    this.panel2 = new System.Windows.Forms.Panel();
    this.pictureBox1 = new System.Windows.Forms.PictureBox();
    this.groupBox1 = new System.Windows.Forms.GroupBox();
    this.textBox1 = new System.Windows.Forms.TextBox();
    this.textBox3 = new System.Windows.Forms.TextBox();
    this.button4 = new System.Windows.Forms.Button();
    this.tabPage2 = new System.Windows.Forms.TabPage();
    this.tabPage1 = new System.Windows.Forms.TabPage();
    this.tabControl1 = new System.Windows.Forms.TabControl();
    this.button3 = new System.Windows.Forms.Button();
    this.textBox2 = new System.Windows.Forms.TextBox();
    this.panel1.SuspendLayout();
    this.panel2.SuspendLayout();
    ((System.ComponentModel.ISupportInitialize)(this.pictureBox1)).BeginInit();
    this.groupBox1.SuspendLayout();
    this.tabPage2.SuspendLayout();
    this.tabControl1.SuspendLayout();
    this.SuspendLayout();
    //
    // label1
    //
    this.label1.BackColor = System.Drawing.Color.LightGreen;
    this.label1.Location = new System.Drawing.Point(24, 16);
    this.label1.Name = "label1";
    this.label1.Size = new System.Drawing.Size(131, 17);
    this.label1.TabIndex = 0;
    this.label1.Text = "label1";
    // les lignes précédentes encadrées affichent ceci :
```

label1

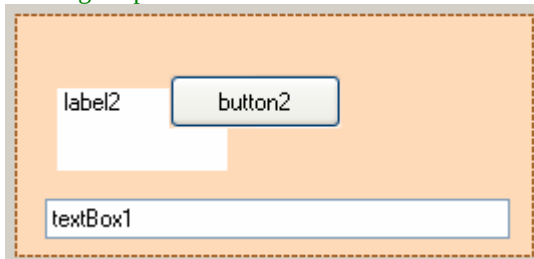
```
//
// panel1
//
this.panel1.BackColor = System.Drawing.Color.PeachPuff;
this.panel1.Controls.Add(this.textBox1);
this.panel1.Controls.Add(this.button2);
this.panel1.Controls.Add(this.label2);
this.panel1.Location = new System.Drawing.Point(12, 65);
this.panel1.Name = "panel1";
this.panel1.Size = new System.Drawing.Size(259, 121);
this.panel1.TabIndex = 1;
//
// button2
//
```

```

this.button2.Location = new System.Drawing.Point(77, 30);
this.button2.Name = "button2";
this.button2.Size = new System.Drawing.Size(86, 27);
this.button2.TabIndex = 1;
this.button2.Text = "button2";
this.button2.UseVisualStyleBackColor = true;
//
// label2
//
this.label2.BackColor = System.Drawing.Color.White;
this.label2.Location = new System.Drawing.Point(21, 37);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(85, 41);
this.label2.TabIndex = 0;
this.label2.Text = "label2";
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(16, 76);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(232, 20);
this.textBox1.TabIndex = 2;
this.textBox1.Text = "textBox1";

```

// les lignes précédentes encadrées affichent ceci :



```

//
// button1
//
this.button1.BackColor = System.Drawing.Color.PaleTurquoise;
this.button1.Location = new System.Drawing.Point(29, 29);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(86, 27);
this.button1.TabIndex = 1;
this.button1.Text = "button1";
this.button1.UseVisualStyleBackColor = false;
//
// panel2
//
this.panel2.BackColor = System.Drawing.Color.PaleTurquoise;
this.panel2.Controls.Add(this.pictureBox1);
this.panel2.Controls.Add(this.button1);
this.panel2.Location = new System.Drawing.Point(230, 27);
this.panel2.Name = "panel2";
this.panel2.Size = new System.Drawing.Size(225, 144);
this.panel2.TabIndex = 2;
//
// pictureBox1
//
this.pictureBox1.Image = global::ProjApplication0.Properties.Resources.net20;
this.pictureBox1.Location = new System.Drawing.Point(87, 68);
this.pictureBox1.Name = "pictureBox1";
this.pictureBox1.Size = new System.Drawing.Size(109, 48);

```

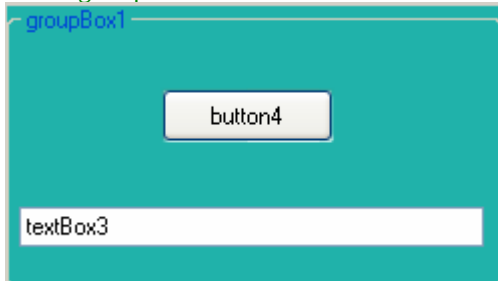
```
this.pictureBox1.SizeMode = System.Windows.Forms.PictureBoxSizeMode.AutoSize;
this.pictureBox1.TabIndex = 2;
this.pictureBox1.TabStop = false;
```

// les lignes précédentes encadrées affichent ceci :



```
//
// groupBox1
//
this.groupBox1.BackColor = System.Drawing.Color.LightSeaGreen;
this.groupBox1.Controls.Add(this.textBox3);
this.groupBox1.Controls.Add(this.button4);
this.groupBox1.Location = new System.Drawing.Point(12, 216);
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size(247, 140);
this.groupBox1.TabIndex = 3;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "groupBox1";
//
// textBox3
//
this.textBox3.Location = new System.Drawing.Point(6, 100);
this.textBox3.Name = "textBox3";
this.textBox3.Size = new System.Drawing.Size(232, 20);
this.textBox3.TabIndex = 4;
this.textBox3.Text = "textBox3";
//
// button4
//
this.button4.Location = new System.Drawing.Point(77, 41);
this.button4.Name = "button4";
this.button4.Size = new System.Drawing.Size(86, 27);
this.button4.TabIndex = 5;
this.button4.Text = "button4";
this.button4.UseVisualStyleBackColor = true;
```

// les lignes précédentes encadrées affichent ceci :



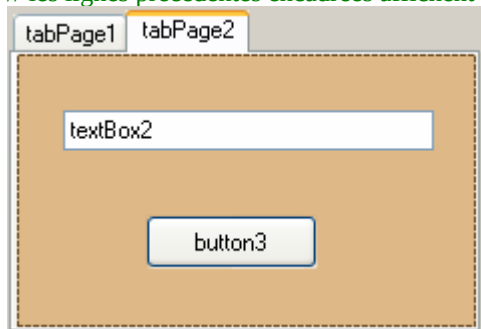
```
//
// tabPage2
//
this.tabPage2.BackColor = System.Drawing.Color.BurlyWood;
this.tabPage2.Controls.Add(this.button3);
```

```

this.tabPage2.Controls.Add(this.textBox2);
this.tabPage2.Location = new System.Drawing.Point(4, 22);
this.tabPage2.Name = "tabPage2";
this.tabPage2.Padding = new System.Windows.Forms.Padding(3);
this.tabPage2.Size = new System.Drawing.Size(229, 136);
this.tabPage2.TabIndex = 1;
this.tabPage2.Text = "tabPage2";
//
// tabPage1
//
this.tabPage1.BackColor = System.Drawing.Color.MediumAquamarine;
this.tabPage1.Location = new System.Drawing.Point(4, 22);
this.tabPage1.Name = "tabPage1";
this.tabPage1.Padding = new System.Windows.Forms.Padding(3);
this.tabPage1.Size = new System.Drawing.Size(229, 136);
this.tabPage1.TabIndex = 0;
this.tabPage1.Text = "tabPage1";
this.tabPage1.UseVisualStyleBackColor = true;
//
// tabControl1
//
this.tabControl1.Controls.Add(this.tabPage1);
this.tabControl1.Controls.Add(this.tabPage2);
this.tabControl1.HotTrack = true;
this.tabControl1.Location = new System.Drawing.Point(272, 207);
this.tabControl1.Name = "tabControl1";
this.tabControl1.SelectedIndex = 0;
this.tabControl1.Size = new System.Drawing.Size(237, 162);
this.tabControl1.TabIndex = 4;
//
// button3
//
this.button3.Location = new System.Drawing.Point(64, 80);
this.button3.Name = "button3";
this.button3.Size = new System.Drawing.Size(86, 27);
this.button3.TabIndex = 4;
this.button3.Text = "button3";
this.button3.UseVisualStyleBackColor = true;
//
// textBox2
//
this.textBox2.Location = new System.Drawing.Point(23, 28);
this.textBox2.Name = "textBox2";
this.textBox2.Size = new System.Drawing.Size(185, 20);
this.textBox2.TabIndex = 5;
this.textBox2.Text = "textBox2";

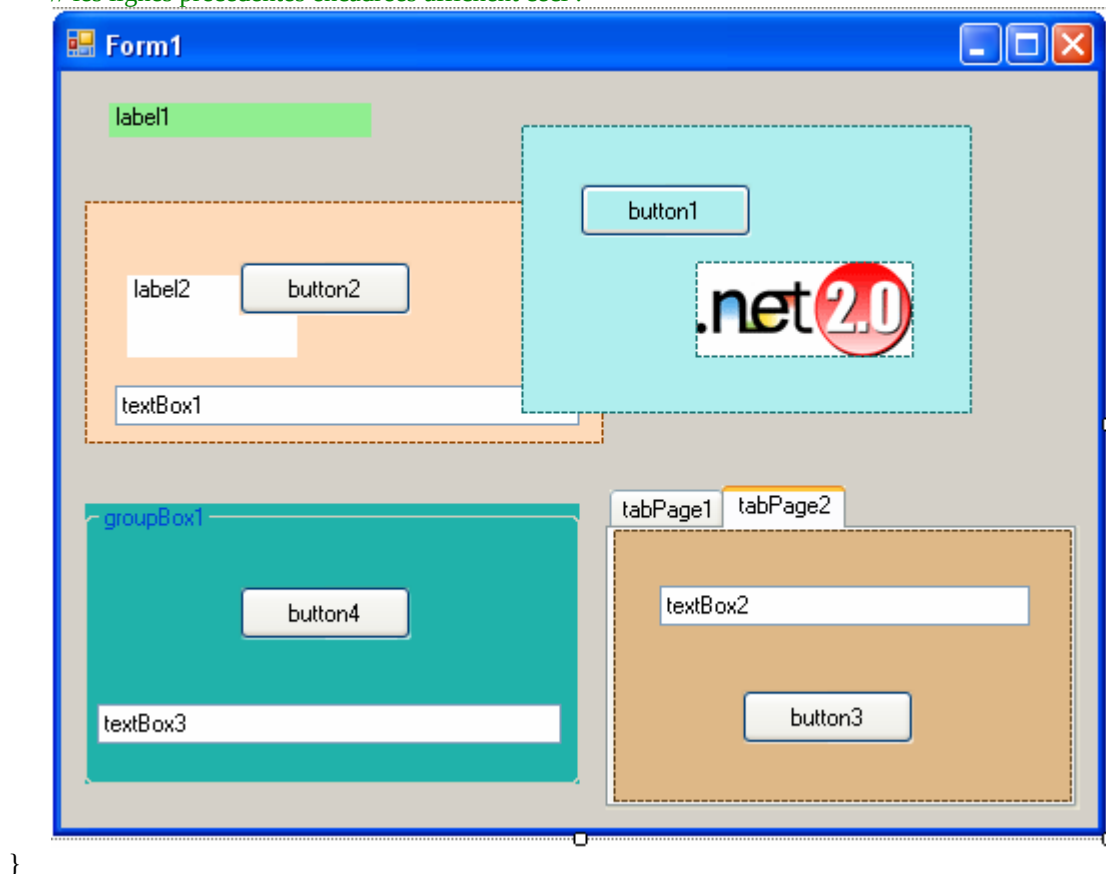
```

// les lignes précédentes encadrées affichent ceci :



```
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.BackColor = System.Drawing.SystemColors.ActiveBorder;
this.ClientSize = new System.Drawing.Size(518, 378);
this.Controls.Add(this.tabControl1);
this.Controls.Add(this.groupBox1);
this.Controls.Add(this.panel2);
this.Controls.Add(this.panel1);
this.Controls.Add(this.label1);
this.Name = "Form1";
this.Text = "Form1";
this.panel1.ResumeLayout(false);
this.panel1.PerformLayout();
this.panel2.ResumeLayout(false);
this.panel2.PerformLayout();
((System.ComponentModel.ISupportInitialize)(this.pictureBox1)).EndInit();
this.groupBox1.ResumeLayout(false);
this.groupBox1.PerformLayout();
this.tabPage2.ResumeLayout(false);
this.tabPage2.PerformLayout();
this.tabControl1.ResumeLayout(false);
this.ResumeLayout(false);
```

// les lignes précédentes encadrées affichent ceci :



#endregion

```
private System.Windows.Forms.Label label1;
private System.Windows.Forms.Panel panel1;
```

```

private System.Windows.Forms.Button button2;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.Panel panel2;
private System.Windows.Forms.PictureBox pictureBox1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.GroupBox groupBox1;
private System.Windows.Forms.TextBox textBox3;
private System.Windows.Forms.Button button4;
private System.Windows.Forms.TabPage tabPage2;
private System.Windows.Forms.Button button3;
private System.Windows.Forms.TextBox textBox2;
private System.Windows.Forms.TabPage tabPage1;
private System.Windows.Forms.TabControl tabControl1;

```

Déclaration des références des objets
de type composants

```

}
}

```

1.3 Influence de la propriété parent sur l'affichage visuel d'un contrôle

Dans l'IHM précédente, programmons par exemple la modification de la propriété Parent du contrôle textBox1 en réaction au click de souris sur les Button button1 et button2.

Il faut abonner le gestionnaire du click de button1 "**private void** button1_Click", au délégué button1.Click :

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

De même il faut abonner le gestionnaire du click de button2 "**private void** button2_Click", au délégué button2.Click :

```
this.button2.Click += new System.EventHandler(this.button2_Click);
```

Le RAD engendre automatiquement les gestionnaires:

```

private void button1_Click ( object sender, System.EventArgs e ){ }
private void button1_Click ( object sender, System.EventArgs e ){ }

```

Les lignes d'abonnement sont engendrées dans la méthode InitializeComponent () :

```

private void InitializeComponent ()
{ ....

this.button1.Click += new System.EventHandler(this.button1_Click);
this.button2.Click += new System.EventHandler(this.button2_Click);
}

```

Le code et les affichages obtenus (le textBox1 est positionné en X=16 et Y=76 sur son parent) :

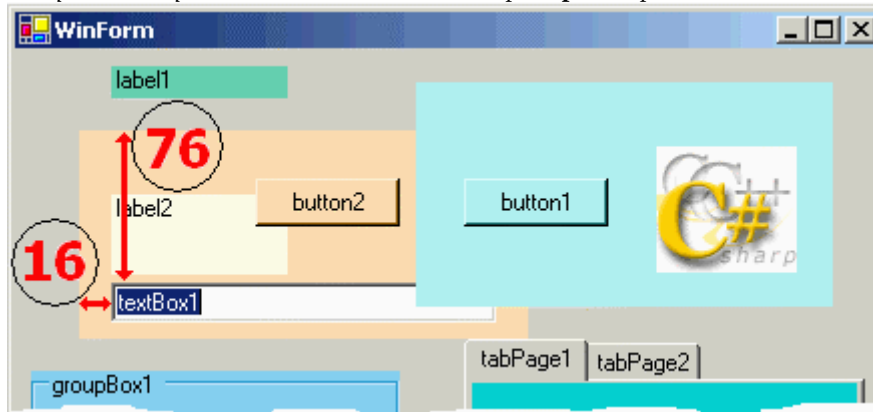
// Gestionnaire du click sur button1 :

```
private void button1_Click ( object sender, System.EventArgs e ){  
    textBox1.Parent = panel1 ;  
}
```

Le composant a déjà été positionné par l'instruction suivante :

```
this.textBox1.Location = new System.Drawing.Point(16, 76);
```

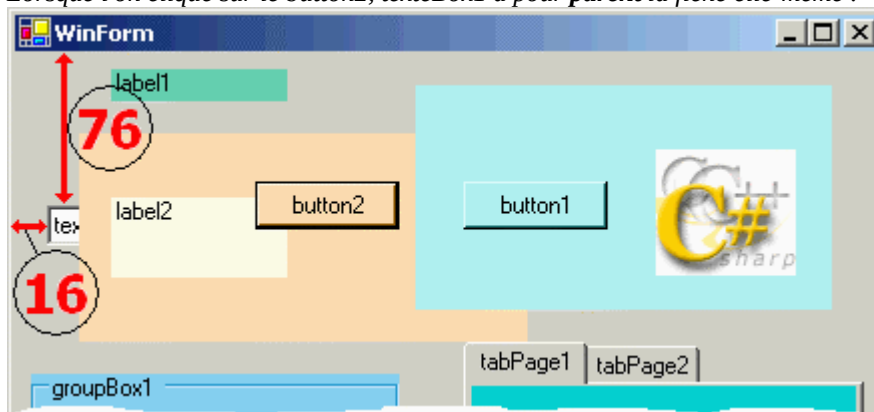
Lorsque l'on clique sur le button1, textBox1 a pour **parent** panel1 :



// Gestionnaire du click sur button2 :

```
private void button2_Click ( object sender, System.EventArgs e ){  
    textBox1.Parent = this;  
}
```

Lorsque l'on clique sur le button2, textBox1 a pour **parent** la fiche elle-même :



Le contrôle pictureBox1 permet d'afficher des images : ico, bmp, gif, png, jpg, jpeg

// chargement d'un fichier image dans le pictureBox1 par un click sur button3 :

```
private void InitializeComponent ()  
{ ....
```

```

this.button1.Click += new System.EventHandler(this.button1_Click);
this.button2.Click += new System.EventHandler(this.button2_Click);
this.button3.Click += new System.EventHandler(this.button3_Click);
}

```

```

private void button3_Click ( object sender, System.EventArgs e ) {
    pictureBox1.Image = Image.FromFile("csharp.jpg");
}

```

Lorsque l'on clique sur le button3, l'image "csharp.jpg" est chargée dans pictureBox1 :



1.4 Des graphiques dans les formulaires avec le GDI+

Nous avons remarqué que C# possède un contrôle permettant l'affichage d'images de différents formats, qu'en est-il de l'affichage de graphiques construits pendant l'exécution ? Le GDI+ répond à cette question.

Le **Graphical Device Interface+** est la partie de NetFrameWork qui fournit les graphismes vectoriels à deux dimensions, les images et la typographie. GDI+ est une interface de périphérique graphique qui permet aux programmeurs d'écrire des applications indépendantes des périphériques physiques (écran, imprimante,...).

Lorsque l'on dessine avec GDI+, on utilise des méthodes de classes situées dans le GDI+, donnant des directives de dessin, ce sont ces méthodes qui, via le CLR du NetFrameWork, font appel aux pilotes du périphérique physique, les programmes ainsi conçus ne dépendent alors pas du matériel sur lequel ils s'afficheront.

Pour dessiner des graphiques sur n'importe quel périphérique d'affichage, il faut un objet **Graphics**. Un objet de classe **System.Drawing.Graphics** est associé à une surface de dessin, généralement la zone cliente d'un formulaire (objet de classe Form). Il n'y a pas de constructeur dans la classe Graphics :

Graphics Obj = new Graphics();

Impossible

Comme le dessin doit avoir lieu sur la surface visuelle d'un objet visuel donc un contrôle, c'est cet objet visuel qui fournit le fond, le GDI+ fournit dans la classe **System.Windows.Forms.Control**

la méthode **CreateGraphics** qui permet de créer un objet de type **Graphics** qui représente le "fond de dessin" du contrôle :

Bibliothèque de classes .NET Framework Control, méthodes	
 CreateGraphics Pris en charge par le .NET Compact Framework.	Crée l'objet Graphics pour le contrôle.

Syntaxe :

```
public Graphics CreateGraphics( );
```

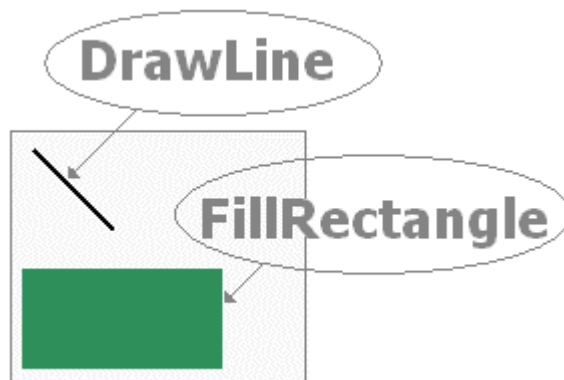
Afin de comprendre comment utiliser un objet Graphics construisons un exemple fictif de code dans lequel on suppose avoir instancié ObjVisuel un contrôle (par exemple : une fiche, un panel,...), on utilise deux méthodes dessin de la classe Graphics pour dessiner un trait et un rectangle :

Bibliothèque de classes .NET Framework Graphics, méthodes	
 DrawLine Pris en charge par le .NET Compact Framework.	Surchargé. Dessine une ligne reliant les deux points spécifiés par des paires de coordonnées.
 FillRectangle Pris en charge par le .NET Compact Framework.	Surchargé. Remplit l'intérieur d'un rectangle spécifié par une paire de coordonnées, une largeur et une hauteur.

Code C#	Explication
Graphics fond = ObjVisuel.CreateGraphics ();	Obtention d'un fond de dessin sur ObjVisuel (création d'un objet Graphics associé à ObjVisuel)
Pen blackPen = new Pen (Color.Black, 2);	Création d'un objet de pinceau de couleur noire et d'épaisseur 2 pixels
fond.DrawLine (blackPen, 10f, 10f, 50f, 50f);	Utilisation du pinceau blackPen pour tracer une ligne droite sur le fond d'ObjVisuel entre les deux points A(10,10) et B(50,50).
fond.FillRectangle (Brushes.SeaGreen,5,70,100,50);	Utilisation d'une couleur de brosse SeaGreen, pour remplir l'intérieur d'un rectangle spécifié par une paire de coordonnées (5,70), une largeur(100 pixels) et une hauteur (50 pixels).

Ces quatre instructions ont permis de dessiner le trait noir et le rectangle vert sur le fond du

contrôle ObjVisuel représenté ci-dessous par un rectangle à fond blanc :



Note technique de Microsoft

L'objet Graphics retourné doit être supprimé par l'intermédiaire d'un appel à sa méthode **Dispose** lorsqu'il n'est plus nécessaire.

La classe Graphics implémente l'interface IDisposable :

public sealed class Graphics : MarshalByRefObject, **IDisposable**

Les objets Graphics peuvent donc être libérés par la méthode Dispose().

Afin de respecter ce conseil d'optimisation de gestion de la mémoire, nous rajoutons dans notre code l'appel à la méthode **Dispose** de l'objet Graphics. Nous prenons comme ObjVisuel un contrôle de type panel que nous nommons panelDessin (**private System.Windows.Forms.Panel panelDessin**):

```
//...
Graphics fond = panelDessin.CreateGraphics ( );
Pen blackPen = new Pen ( Color.Black, 2 );
fond.DrawLine ( blackPen, 10f, 10f, 50f, 50f );
fond.FillRectangle ( Brushes.SeaGreen,5,70,100,50 );
fond.Dispose( );
//...suite du code où l'objet fond n'est plus utilisé
```

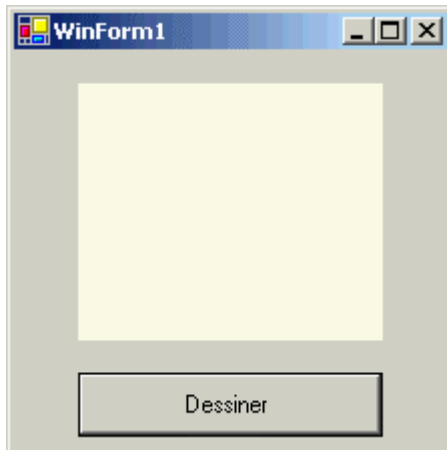
Nous pouvons aussi utiliser l'instruction **using** déjà vue qui libère automatiquement l'objet par appel à sa méthode Dispose :

```
//...
using( Graphics fond = panelDessin.CreateGraphics ( ) ) {
    Pen blackPen = new Pen ( Color.Black, 2 );
    fond.DrawLine ( blackPen, 10f, 10f, 50f, 50f );
    fond.FillRectangle ( Brushes.SeaGreen,5,70,100,50 );
}
//...suite du code où l'objet fond n'est plus utilisé
```

1.5 Le dessin doit être persistant

Reprenons le dessin précédent et affichons-le à la demande.

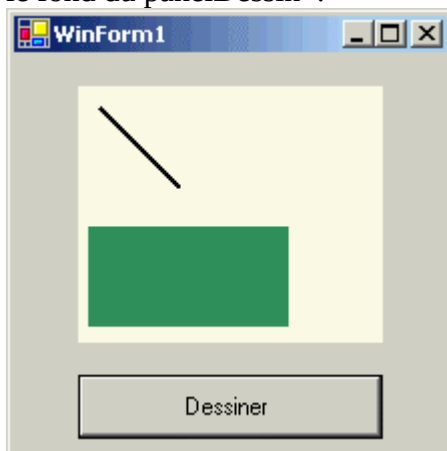
Nous supposons disposer d'un formulaire nommé WinForm1 contenant un panneau nommé panelDessin (**private System.Windows.Forms.Panel** panelDessin) et un bouton nommé buttonDessin (**private System.Windows.Forms.Button** buttonDessin)



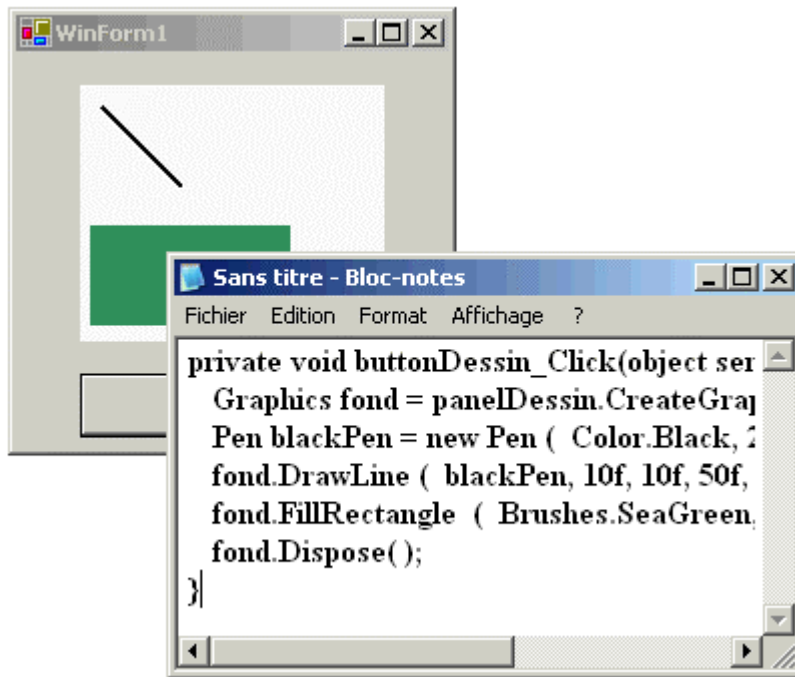
Nous programmons un gestionnaire de l'événement click du buttonDessin, dans lequel nous copions le code de traçage de notre dessin :

```
private void buttonDessin_Click(object sender, System.EventArgs e) {  
    Graphics fond = panelDessin.CreateGraphics ( );  
    Pen blackPen = new Pen ( Color.Black, 2 );  
    fond.DrawLine ( blackPen, 10f, 10f, 50f, 50f );  
    fond.FillRectangle ( Brushes.SeaGreen,5,70,100,50 );  
    fond.Dispose( );  
}
```

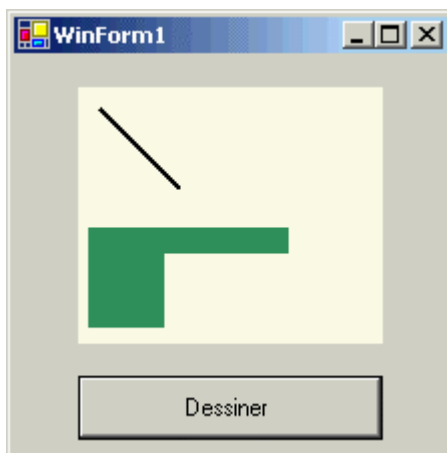
Lorsque nous cliquons sur le bouton buttonDessin, le trait noir et le rectangle vert se dessine sur le fond du panelDessin :



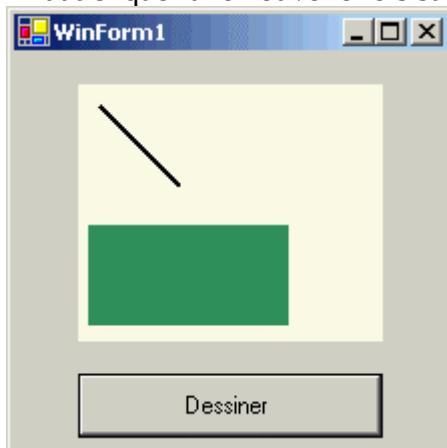
Faisons apparaître une fenêtre de bloc-note contenant du texte, qui masque partiellement notre formulaire WinForm1 qui passe au second plan comme ci-dessous :



Si nous nous refocalisons sur le formulaire en cliquant sur lui par exemple, celui-ci repasse au premier plan, nous constatons que notre dessin est abîmé. Le rectangle vert est amputé de la partie qui était recouverte par la fenêtre de bloc-note. Le formulaire s'est bien redessiné, mais pas nos tracés ;



Il faut cliquer une nouvelle fois sur le bouton pour lancer le redessin des tracés :



Il existe un moyen simple permettant d'effectuer le redessinement de nos tracés lorsque le formulaire se redessine lui-même automatiquement : il nous faut "consommer" l'événement **Paint** du formulaire qui se produit lorsque le formulaire est redessiné (ceci est d'ailleurs valable pour n'importe quel contrôle). La consommation de l'événement Paint s'effectue grâce au gestionnaire Paint de notre formulaire WinForm1 :

```
private void WinForm1_Paint(object sender, System.Windows.Forms.PaintEventArgs e) {  
    Graphics fond = panelDessin.CreateGraphics();  
    Pen blackPen = new Pen(Color.Black, 2);  
    fond.DrawLine(blackPen, 10f, 10f, 50f, 50f);  
    fond.FillRectangle(Brushes.SeaGreen, 5, 70, 100, 50);  
    fond.Dispose();  
}
```

Le RAD a enregistré (abonné) le gestionnaire **WinForm1_Paint** auprès du délégué **Paint** dans le corps de la méthode InitializeComponent :

```
private void InitializeComponent() {  
    ...  
    this.Paint += new System.Windows.Forms.PaintEventHandler(this.WinForm1_Paint);  
    ...  
}
```

1.6 Deux exemples de graphiques sur plusieurs contrôles

Premier exemple : une méthode générale pour tout redessiner.

Il est possible de dessiner sur tous les types de conteneurs visuels ci-dessous un formulaire nommé Form1 et deux panel (panel1 : jaune foncé et panel2 : violet), la label1 ne sert qu'à afficher du texte en sortie :



Nous écrivons une méthode **TracerDessin** permettant de dessiner sur le fond de la fiche, sur le fond des deux panel et d'écrire du texte sur le fond d'un panel. La méthode **TracerDessin** est appelée dans le gestionnaire de l'événement Paint du formulaire lors de son redessinement de la

fiche afin d'assurer la persistance de tous les tracés. Voici le code qui est créé :

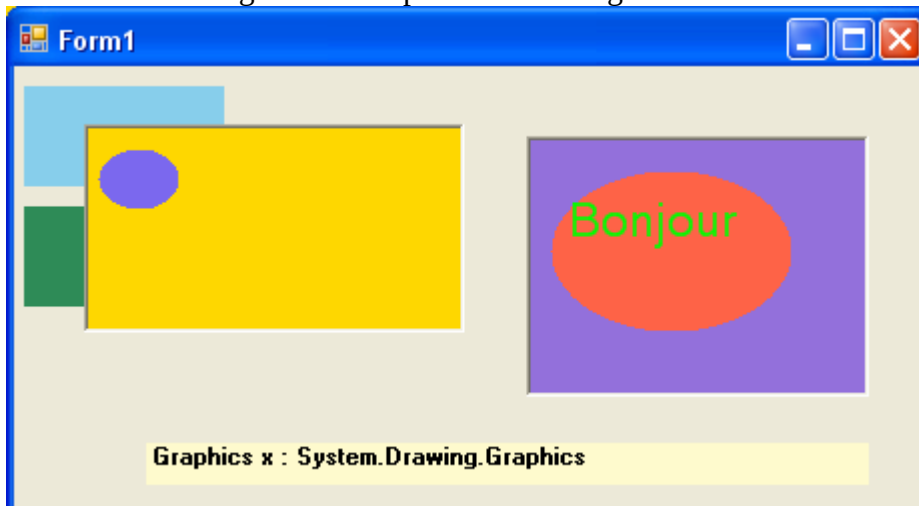
Automatiquement généré dans Form1.Designer.cs :

```
private void InitializeComponent() {  
    ...  
    this.Paint += new System.Windows.Forms.PaintEventHandler ( this Form1_Paint );  
    ...  
}
```

Ecrit dans le gestionnaire de l'événement Paint de Form1, dans Form1.cs :

```
private void Form1_Paint (object sender, System.Windows.Forms.PaintEventArgs e) {  
  
    TracerDessin ( e.Graphics );  
  
    /* Explications sur l'appel de la méthode TracerDessin :  
    Le paramètre e de type PaintEventArgs contient les données relatives à l'événement Paint  
    en particulier une propriété Graphics qui renvoie le graphique utilisé pour peindre sur la fiche  
    c'est pourquoi e.Graphics est passé comme fond en paramètre à notre méthode de dessin.  
    */  
}  
  
private void TracerDessin ( Graphics x ){  
    string Hdcontext ;  
    Hdcontext = x.GetType () .ToString () ;  
    label1.Text = "Graphics x : " + Hdcontext.ToString () ;  
    x.FillRectangle ( Brushes.SeaGreen,5,70,100,50 );  
  
    using( Graphics g = this.CreateGraphics () ) {  
        g.FillRectangle ( Brushes.SkyBlue,5,10,100,50 );  
    }  
    using( Graphics g = panel1.CreateGraphics () ) {  
        g.FillEllipse ( Brushes.MediumSlateBlue,5,10,40,30 );  
    }  
    using( Graphics h = panel2.CreateGraphics () ) {  
        h.FillEllipse ( Brushes.Tomato,10,15,120,80 );  
        h.DrawString ("Bonjour" , new Font (this.Font.FontFamily.Name,18 ) ,Brushes.Lime,15,25 );  
    }  
}
```

L'exécution de ce gestionnaire produit l'affichage suivant :



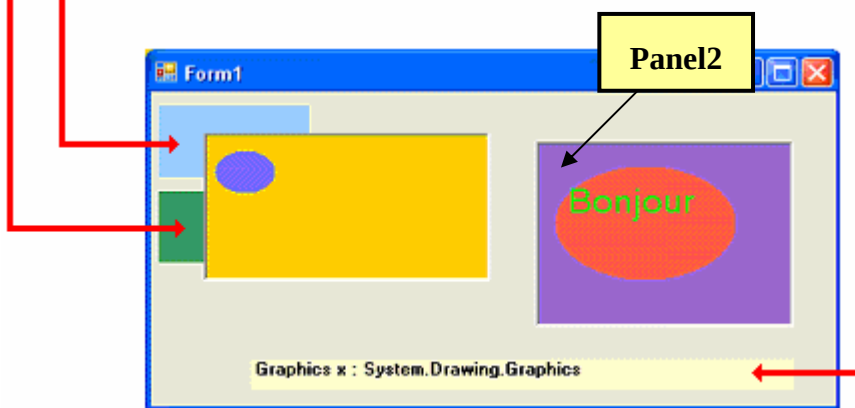
Illustrons les actions de dessin de chaque ligne de code de la méthode **TracerDessin** :

```
private void TracerDessin ( Graphics x ) {
```

```
    string Hdcontext;  
    Hdcontext = x.GetType().ToString();  
    label1.Text = "Graphics x : " + Hdcontext.ToString();
```

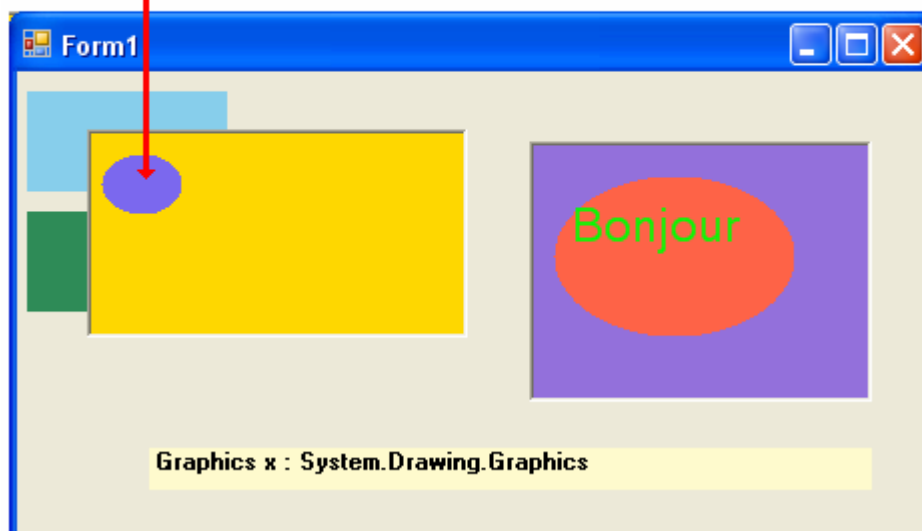
```
    x.FillRectangle ( Brushes.SeaGreen,5,70,100,50);
```

```
    using( Graphics g = this.CreateGraphics() ) {  
        g.FillRectangle ( Brushes.SkyBlue,5,10,100,50);  
    }
```



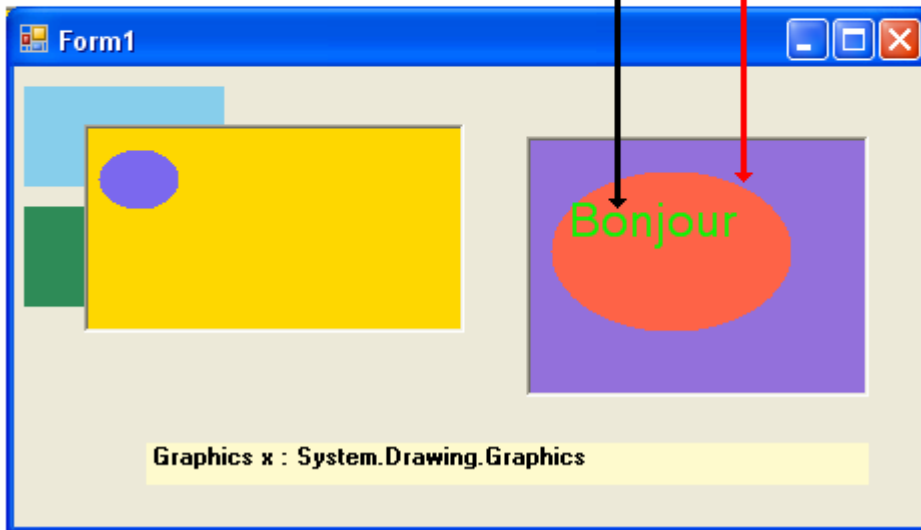
On dessine deux rectangles sur le fond de la fiche, nous notons que ces deux rectangles ont une intersection non vide avec le panel2 (jaune foncé) et que cela n'altère pas le dessin du panel. En effet le panel est un contrôle et donc se redessine lui-même. Nous en déduisons que le fond graphique est situé "en dessous" du dessin des contrôles.

```
    using( Graphics g = panel1.CreateGraphics() ) {  
        g.FillEllipse ( Brushes.MediumSlateBlue,5,10,40,30);  
    }
```



L'instruction dessine l'ellipse bleue à gauche sur le fond du panel1

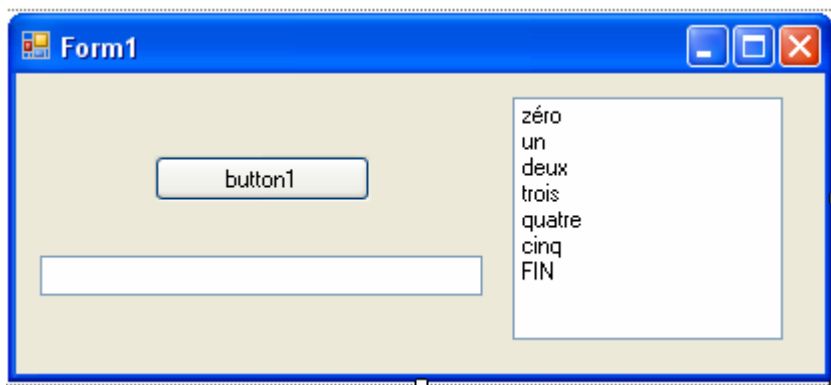
```
using( Graphics h = panel2.CreateGraphics() ) {  
    h.FillEllipse ( Brushes.Tomato,10,15,120,80 );  
    h.DrawString("Bonjour", new Font(this.Font.FontFamily.Name,18 ),Brushes.Lime,15,25 );  
}
```



La première instruction dessine l'ellipse rouge, la seconde écrit le texte "Bonjour" en vert sur le fond du panel2.

Deuxième exemple : traçé des dessins sur des événements spécifiques

Le deuxième exemple montre que l'on peut dessiner aussi sur le fond d'autres contrôles différents des contrôles conteneurs; nous dessinons deux rectangles vert et bleu sur le fond du formulaire et un petit rectangle bleu ciel dans un ListBox, un TextBox et Button déposés sur le formulaire :



Le code de Form1.Designer.cs est le suivant :

```

namespace WindowsApplication3
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.listBox1 = new System.Windows.Forms.ListBox();
            this.textBox1 = new System.Windows.Forms.TextBox();
            this.SuspendLayout();
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(69, 41);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(108, 23);
            this.button1.TabIndex = 0;
            this.button1.Text = "button1";
            this.button1.UseVisualStyleBackColor = true;
            //
            // listBox1
            //
            this.listBox1.FormattingEnabled = true;
            this.listBox1.Items.AddRange(new object[] {
                "zéro",
                "un",
                "deux",
                "trois",
                "quatre",
                "cinq",
                "FIN"});
            this.listBox1.Location = new System.Drawing.Point(248, 12);
            this.listBox1.Name = "listBox1";
            this.listBox1.Size = new System.Drawing.Size(135, 121);
            this.listBox1.TabIndex = 1;
            //
            // textBox1

```

```

//
this.textBox1.Location = new System.Drawing.Point(12, 91);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(221, 20);
this.textBox1.TabIndex = 2;
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(404, 150);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.listBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);
this.PerformLayout();

}

#endregion

private System.Windows.Forms.Button button1;
private System.Windows.Forms.ListBox listBox1;
private System.Windows.Forms.TextBox textBox1;
}
}

```

Les graphiques sont gérés dans Form1.cs avec le code ci-après :

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication3
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        //-- dessin persistant sur le fond de la fiche par gestionnaire Paint:
        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            g.FillRectangle(Brushes.SeaGreen, 5, 70, 100, 50);
            g.FillRectangle(Brushes.SkyBlue, 5, 10, 100, 50);
        }
        //-- dessin persistant sur le fond du bouton par gestionnaire Paint:
        private void button1_Paint(object sender, PaintEventArgs e)
        {
            Graphics x = e.Graphics;
            x.FillRectangle(Brushes.SkyBlue, 5, 5, 20, 20);
        }
    }
}

```

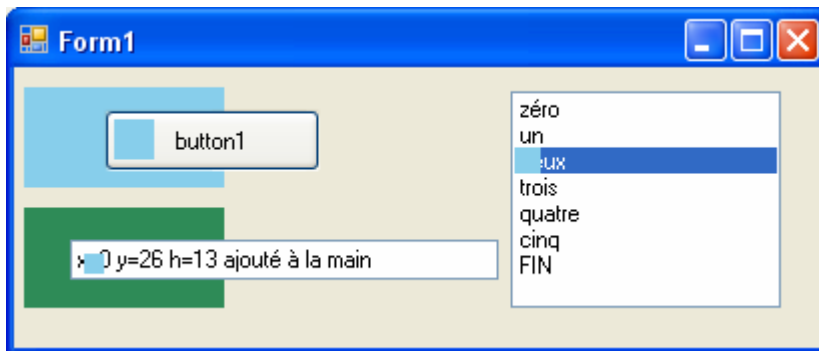
```

//-- dessin non persistant sur le fond du ListBox par gestionnaire SelectedIndexChanged :
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    Rectangle Rect = listBox1.GetItemRectangle(listBox1.SelectedIndex);
    int Haut = listBox1.ItemHeight;
    textBox1.Text = "x=" + Rect.X.ToString() + " y=" + Rect.Y.ToString() + " h=" + Haut.ToString();
    using (Graphics k = listBox1.CreateGraphics())
    {
        k.FillRectangle(Brushes.SkyBlue, Rect.X, Rect.Y, Haut, Haut);
    }
}

//-- dessin non persistant sur le fond du TextBox par gestionnaire TextChanged :
private void textBox1_TextChanged(object sender, EventArgs e)
{
    using (Graphics k = textBox1.CreateGraphics())
    {
        k.FillRectangle(Brushes.SkyBlue, 5, 5, 10, 10);
    }
}
}
}

```

Après exécution, sélection de la 3ème ligne de la liste et ajout d'un texte au clavier dans le TextBox :



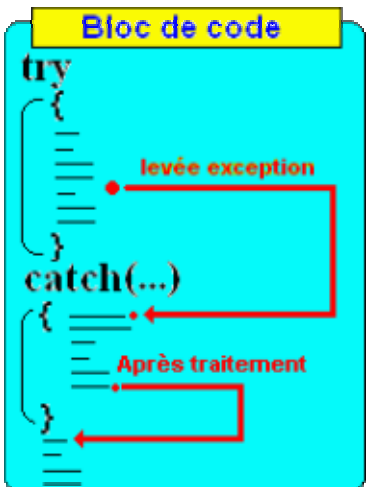
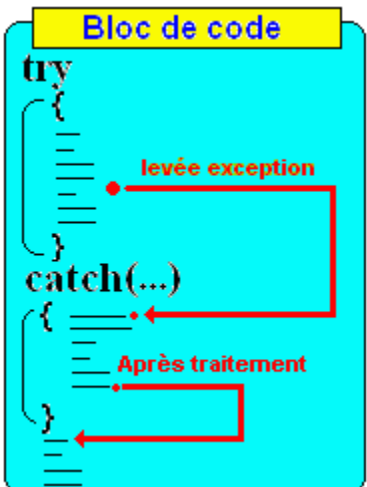
Exceptions : C# comparé à Java et Delphi



1. similitude et différence

Le langage C# hérite strictement de Java pour la syntaxe et le fonctionnement de base des exceptions et de la simplicité de Delphi dans les types d'exceptions.

Pour une étude complète de la notion d'exception, de leur gestion, de la hiérarchie, de l'ordre d'interception et du redéclenchement d'une exception, nous renvoyons le lecteur au chapitre Traitement d'exceptions du présent ouvrage. Nous figurons ci-dessous un tableau récapitulant les similitudes dans chacun des trois langages :

Delphi	Java	C#
<pre>try - ... <lignes de code à protéger> - ... except on E : ExxException do begin - ... <lignes de code réagissant à l'exception> - ... end; - ... end ;</pre>	<pre>try { - ... <lignes de code à protéger> - ... } catch (ExxException E) { - ... <lignes de code réagissant à l'exception> - ... }</pre> fonctionnement identique à C#	<pre>try { - ... <lignes de code à protéger> - ... } catch (ExxException E) { - ... <lignes de code réagissant à l'exception> - ... }</pre> fonctionnement identique à Java
		

ATTENTION DIFFERENCE : C# - Java

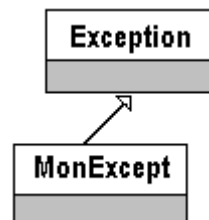
Seul Java possède deux catégories d'exceptions : les exceptions **vérifiées** et les exceptions **non vérifiées**. C# comme Delphi possède un mécanisme plus simple qui est équivalent à celui de Java dans le cas des exceptions **non vérifiées (implicites)** (la propagation de l'exception est **implicitement** prise en charge par le système d'exécution).

2. Créer et lancer ses propres exceptions

Dans les 3 langages la classes de base des exceptions se nomme : **Exception**

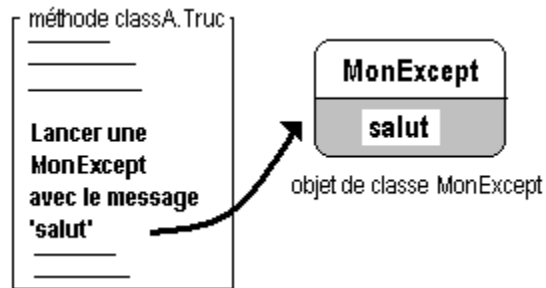
- Il est possible de construire de nouvelles classes d'exceptions personnalisées en héritant d'une des classes du langage, ou à minima de la classe de base **Exception**.
- Il est aussi possible de lancer une exception personnalisée (comme si c'était une exception propre au langage) à n'importe quel endroit dans le code d'une méthode d'une classe, en instanciant un objet d'exception personnalisé et en le préfixant par le mot clef (**raise** pour Delphi et **throw** pour Java et C#).
- Le mécanisme général d'interception des exceptions à travers des gestionnaires d'exceptions **try...except** ou **try...catch** s'applique à tous les types d'exceptions y compris les exceptions personnalisées.

1°) Création d'une nouvelle classe :



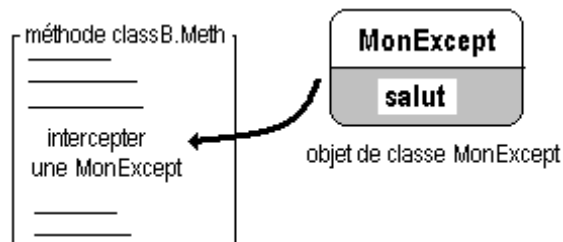
Delphi	Java	C#
Type MonExcept = class (Exception) End; MonExcept hérite par construction de la propriété public Message de sa mère, en lecture et écriture.	class MonExcept extends Exception { public MonExcept (String s) { super (s); } }	class MonExcept : Exception { public MonExcept (string s) : base (s) { } } MonExcept hérite par construction de la propriété public Message de sa mère, en lecture seulement.

2°) Lancer une MonExcept :



Delphi	Java	C#
Type MonExcept = class (Exception) End; classA = class Procedure Truc; end; Implementation Procedure classA.Truc; begin raise MonExcept.Create ('salut'); end;	class MonExcept extends Exception { public MonExcept (String s) { super (s); } } class classA { void Truc () throws MonExcept { throw new MonExcept ('salut'); } }	class MonExcept : Exception { public MonExcept (string s) : base (s) { } } class classA { void Truc () { throw new MonExcept ('salut'); } }

3°) Intercepter une MonExcept :



Delphi	Java	C#
Type MonExcept = class (Exception) End; classB = class Procedure Meth; end; Implementation	class MonExcept extends Exception { public MonExcept (String s) { super (s); } } class classB { void Meth () {	class MonExcept : Exception { public MonExcept (string s) : base (s) { } } class classB { void Meth () {

```
Procedure classB.Meth;
```

```
begin
```

```
.....
```

```
try
```

```
..... // code à protéger
```

```
except on MonExcept do
```

```
begin
```

```
..... // code de réaction à l'exception
```

```
end;
```

```
end;
```

```
end;
```

```
.....
```

```
try
```

```
{
```

```
..... // code à protéger
```

```
}
```

```
catch (MonExcept e) {
```

```
..... // code de réaction à l'exception
```

```
}
```

```
}
```

```
.....
```

```
try
```

```
{
```

```
..... // code à protéger
```

```
}
```

```
catch (MonExcept e) {
```

```
..... // code de réaction à l'exception
```

```
}
```

```
}
```

Données simples et fichiers



Plan général:

1. Les manipulations disque de fichiers et de répertoires

- 1.1 La manipulation des répertoires
- 1.2 La manipulation des chemins d'accès disque
- 1.3 La manipulation des fichiers

2. Flux et fichiers dans NetFramework

- 2.1 Les flux avec C#
- 2.2 Les flux et les fichiers de caractères avec C#
- 2.3 Création d'un fichier de texte avec des données
- 2.4 Copie des données d'un fichier texte dans un autre

3. Exercice de fichier de salariés avec une IHM

Différences entre flux et fichiers

- Un **fichier** est une ensemble de données structurées possédant un nom (le nom du fichier) stockées généralement sur disque (dans une mémoire de masse en général) dans des enregistrement sous forme d'octets. Les fichiers sont en écriture ou en lecture et ont des organisations diverses comme l'accès séquentiel, l'accès direct, l'accès indexé, en outre les fichiers sont organisés sur le disque dans des répertoires nommés selon des chemins d'accès.
- Un **flux** est un tube de connexion entre deux entrepôts pouvant contenir des données. Ces entités entreposant les données peuvent être des mémoire de masse (disques) ou des zones différentes de la mémoire centrale, ou bien des réseaux différents (par exemple échanges de données entre sockets).
- En C#, il faut utiliser un flux pour accéder aux données d'un fichier stocké sur disque, ce flux connecte les données du fichier sur le disque à une zone précise de la mémoire : le tampon mémoire.

Le Framework.Net dispose d'une famille de classes permettant la manipulation de données externes : **System.IO**

Espace de noms **System.IO**

- contient des classes qui permettent la lecture et l'écriture dans des fichiers,
- contient des classes qui permettent la lecture et l'écriture dans des flux ,
- contient des classes qui permettent la création, le renommage, le déplacement de fichiers et de répertoires sur disque.

1. Les manipulations disque de fichiers et de répertoires

L'espace de nom **System.IO** contient plusieurs classes de manipulation des fichiers et des répertoires. Certaines ne contiennent que des méthodes **static** (rassemblement de méthodes utiles à la manipulation de fichiers ou de répertoires quelconques), d'autres contiennent des méthodes d'instances et sont utiles lorsque l'on veut travailler sur un objet de fichier précis.

System.IO.Directory
System.IO.File
System.IO.Path

System.IO.DirectoryInfo
System.IO.FileInfo

1.1 La manipulation des répertoires

La classe **System.IO.Directory** hérite directement de **System.Object** et ne rajoute que des méthodes **static** pour la création, la suppression, le déplacement et le positionnement de répertoires :

CreateDirectory Delete Exists GetAccessControl GetCreationTime GetCreationTimeUtc GetCurrentDirectory GetDirectories GetDirectoryRoot	GetFiles GetFileSystemEntries GetLastAccessTime GetLastAccessTimeUtc GetLastWriteTime GetLastWriteTimeUtc GetLogicalDrives GetParent Move	SetAccessControl SetCreationTime SetCreationTimeUtc SetCurrentDirectory SetLastAccessTime SetLastAccessTimeUtc SetLastWriteTime SetLastWriteTimeUtc
--	--	--

a) Création d'un répertoire sur disque, s'il n'existe pas au préalable :

```
string cheminRep = @"D:\MonRepertoire";  
if ( !Directory.Exists( cheminRep ) ) {  
    Directory.CreateDirectory( cheminRep );  
}
```

Ces 3 lignes de code créent un répertoire **MonRepertoire** à la racine du disque **D**.

b) Suppression d'un répertoire et de tous ses sous-répertoires sur disque :

```
string cheminRep = @"D:\MonRepertoire";  
if ( Directory.Exists( cheminRep ) ) {  
    Directory.Delete( cheminRep, true );  
}
```

Ces 3 lignes de code suppriment sur le disque D, le répertoire **MonRepertoire** et tous ses sous-répertoires.

c) Déplacement d'un répertoire et de tous ses sous-répertoires sur disque :

```
string cheminRep = "D:\\MonRepertoire";  
string cheminDest = "C:\\NouveauRepertoire";  
if ( Directory.Exists( cheminRep ) ) {  
    Directory.Move( cheminRep, cheminDest );  
}
```

Ces 3 lignes de code déplacent le répertoire **MonRepertoire** du disque D vers le disque C sous le nouveau nom **NouveauRepertoire**.

d) Accès au répertoire disque de travail en cours de l'application :

```
string cheminRepAppli = Directory.GetCurrentDirectory( );
```

Attention, ce répertoire est le dernier répertoire en cours utilisé par l'application, il n'est pas nécessairement celui du processus qui a lancé l'application, ce dernier peut être obtenu à partir de la classe **Application** par la propriété "**static string** StartupPath" :

```
string cheminLancerAppli = Application.StartupPath ;
```

La classe **System.IO.DirectoryInfo** qui hérite directement de **System.Object** sert aussi à la création, à la suppression, au déplacement et au positionnement de répertoires, par rapport à la classe **Directory**, certaines méthodes sont transformées en propriétés afin d'en avoir une utilisation plus souple. Conseil de Microsoft : "*si vous souhaitez réutiliser un objet plusieurs fois, l'utilisation de la méthode d'instance de **DirectoryInfo** à la place des méthodes **static** correspondantes de la classe **Directory** peut être préférable*". Le lecteur choisira selon l'application qu'il développe la classe qui lui procure le plus de confort. Afin de bien comparer ces deux classes nous reprenons avec la classe **DirectoryInfo**, les mêmes exemples de base de la classe **Directory**.

a) Création d'un répertoire sur disque, s'il n'existe pas au préalable :

```
string cheminRep = @"D:\MonRepertoire";  
DirectoryInfo Repertoire = new DirectoryInfo( cheminRep );
```

```
if ( !Repertoire.Exists ) {  
    Repertoire.Create( );  
}
```

Ces lignes de code créent un répertoire **MonRepertoire** à la racine du disque **D**.

b) Suppression d'un répertoire et de tous ses sous-répertoires sur disque :

```
string cheminRep = "D:\\MonRepertoire";  
DirectoryInfo Repertoire = new DirectoryInfo( cheminRep );
```

```
if ( Repertoire.Exists ) {  
    Repertoire.Delete( true );  
}
```

Ces lignes de code suppriment sur le disque **D**, le répertoire **MonRepertoire** et tous ses sous-répertoires.

c) Déplacement d'un répertoire et de tous ses sous-répertoires sur disque :

```
string cheminRep = @"D:\MonRepertoire";  
string cheminDest = @"C:\NouveauRepertoire";
```

```
DirectoryInfo Repertoire = new DirectoryInfo( cheminRep );
```

```
if ( Repertoire.Exists ) {
    Repertoire.MoveTo( cheminDest );
}
```

Ces lignes de code déplacent le répertoire **MonRepertoire** du disque D vers le disque C sous le nouveau nom **NouveauRepertoire**.

1.2 La manipulation des chemins d'accès disque

La classe **System.IO.Path** hérite directement de **System.Object** et ne rajoute que des méthodes et des champs **static** pour la manipulation de chaînes **string** contenant des chemins d'accès disque (soit par exemple à partir d'une **string** contenant un chemin comme "c:\rep1\rep2\rep3\appli.exe", extraction du nom de fichier **appli.exe**, du répertoire **c:\rep1\rep2\rep3**, de l'extension **.exe**, etc...).

Ci-dessous nous donnons quelques méthodes **static** pratiques courantes de la classe permettant les manipulations usuelles d'un exemple de chemin d'accès disque sous Windows :

```
string chemin = "C:\\rep1\\rep2\\appli.exe";
```

ou bien

```
string chemin = @"C:\rep1\rep2\appli.exe";
```

Appel de la méthode static sur la string chemin	résultat obtenu après l'appel
System.IO.Path.GetDirectoryName (chemin)	C:\rep1\rep2
System.IO.Path.GetExtension (chemin)	.exe
System.IO.Path.GetFileName (chemin)	appli.exe
System.IO.Path.GetFileNameWithoutExtension (chemin)	appli
System.IO.Path.GetFullPath ("C:\\rep1\\rep2\\..\\appli.exe")	C:\rep1\appli.exe
System.IO.Path.GetPathRoot (chemin)	C:\

1.3 La manipulation des fichiers

La classe **System.IO.File** hérite directement de **System.Object** et ne rajoute que des méthodes **static** pour la manipulation, la lecture et l'écriture de fichiers sur disque :

AppendAll	GetLastAccessTime	SetAccessControl
AppendAllText	GetLastAccessTimeUtc	SetAttributes
AppendText	GetLastWriteTime	SetCreationTime
Copy	GetLastWriteTimeUtc	SetCreationTimeUtc
Create	Move	SetLastAccessTime
CreateText	Open	SetLastAccessTimeUtc
Decrypt	OpenRead	SetLastWriteTime

Delete	OpenText	SetLastWriteTimeUtc
Encrypt	OpenWrite	WriteAll
Exists	ReadAll	WriteAllBytes
GetAccessControl	ReadAllBytes	WriteAllLines
GetAttributes	ReadAllLines	WriteAllText
GetCreationTime	ReadAllText	
GetCreationTimeUtc	Replace	

Exemple de création d'un fichier et de ses répertoires sur le disque en combinant les classes **Directory**, **Path** et **File** :

```
string cheminRep = @"d:\temp\rep1\fichier.txt";
// création d'un répertoire et d'un sous-répertoire s'ils n'existent pas déjà
if ( !Directory.Exists( Path.GetDirectoryName(cheminRep) ) )
    Directory.CreateDirectory( Path.GetDirectoryName(cheminRep) );
```

```
// effacer le fichier s'il existe déjà
if (File.Exists(cheminRep))
    File.Delete(cheminRep);
```

```
// création du fichier " fichier.txt " sur le disque D : dans " D:\temp\rep1"
FileStream fs = File.Create( cheminRep );
```

Les lignes de code C# précédentes créent sur le disque dur **D:** , les répertoires **temp** et **rep1**, puis le **fichier.txt** selon l'architecture disque ci-après :

D:

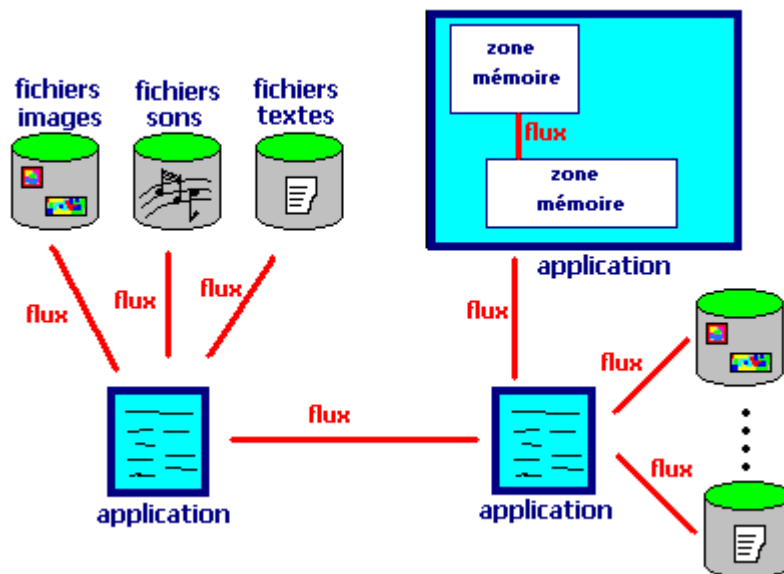


2. Flux et fichiers dans NetFramework

Une application travaille avec ses données internes, mais habituellement il lui est très souvent nécessaire d'aller chercher en **entrée**, on dit **lire**, des nouvelles données (texte, image, son,...) en provenance de diverses sources (périphériques, autres applications, zones mémoires...).

Réciproquement, une application peut après traitement, délivrer en **sortie** des résultats, on dit

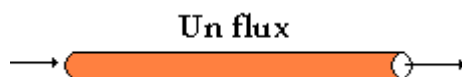
écrire, dans un fichier, vers une autre application ou dans une zone mémoire. **Les flux servent à connecter entre elles ces diverses sources de données.**



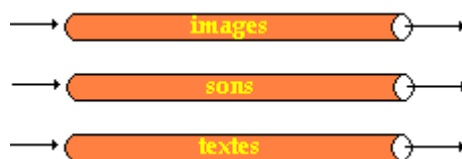
2.1 Les flux avec C#

En C# toutes ces données sont échangées en entrée et en sortie à travers des flux (Stream).

Un flux est une sorte de tuyau de transport séquentiel de données.



Il existe un flux par type de données à transporter :



Un flux est **unidirectionnel** : il y a donc des flux d'**entrée** et des flux de **sortie**.

L'image ci-après montre qu'afin de jouer un son **stocké dans un fichier**, l'application C# ouvre en entrée, un flux associé au fichier de sons et lit ce flux séquentiellement afin ensuite de traiter les sons (modifier ou jouer le son) :



La même application peut aussi traiter des images à partir d'un fichier d'images et renvoyer ces images dans le fichier après traitement. C# ouvre un flux en entrée sur le fichier image et un flux en sortie sur le même fichier, l'application lit séquentiellement le flux d'entrée (octet par octet par exemple) et écrit séquentiellement dans le flux de sortie :



D'une manière générale, NetFramework met à notre disposition dans l'espace de noms **System.IO** une classe abstraite de flux de données considérées comme des séquences d'octets, la classe **System.IO.Stream**. Un certain nombre de classes dérivent de la classe **Stream** et fournissent des implémentations spécifiques en particulier les classes :

Classe NetFramework	Utilisation pratique
System.IO.FileStream	consacrée aux flux connectés sur des fichiers.
System.IO.MemoryStream	consacrée aux flux connectés sur des zones mémoires.
System.Net.Sockets.NetworkStream	consacrée aux flux connectés sur des accès réseau.

Nous allons plus particulièrement étudier quelques exemples d'utilisation de flux connectés sur des fichiers et plus précisément des fichiers de textes.

2.2 Les flux et les fichiers de caractères avec C#

Eu égard à l'importance des fichiers comportant des données textuelles (à bases de caractères) le NetFramework dispose de classe de flux chargés de l'écriture et de la lecture de caractères plus spécialisées que la classe **System.IO.FileStream**.

Les classes de base **abstraites** de ces flux de caractères se dénomment **System.IO.TextReader** pour la classe permettant la création de flux de lecture séquentielle de caractères et **System.IO.TextWriter** pour la classe permettant la création de flux d'écriture séquentielle de caractères.

Les **classes concrètes** et donc pratiques, qui sont fournies par NetFramework et qui implémentent ces deux classes abstraites sont :

System.IO.StreamReader qui dérive et implémente **System.IO.TextReader**.
System.IO.StreamWriter qui dérive et implémente **System.IO.TextWriter**.

Entrées/Sorties synchrone - asynchrone

synchrone

Les accès des flux aux données (lecture par **Read** ou écriture de base par **Write**) sont par défaut en mode **synchrone** c'est à dire : la méthode qui est en train de lire ou d'écrire dans le flux elle ne peut exécuter aucune autre tâche jusqu'à ce que l'opération de lecture ou d'écriture soit achevée. Le traitement des entrées/sorties est alors "**séquentiel**".

asynchrone

Ceci peut être pénalisant si l'application travaille sur de grandes quantités de données et surtout lorsque plusieurs entrées/sorties doivent avoir lieu "en même temps", dans cette éventualité NetFramework fournit des entrées/sorties multi-threadées (qui peuvent avoir lieu "en même temps") avec les méthodes **asynchrones** de lecture **BeginRead** et **EndRead** et d'écriture **BeginWrite** et **EndWrite**). Dans le cas d'utilisation de méthodes asynchrones, le thread principal peut continuer à effectuer d'autres tâches : le traitement des entrées/sorties est alors "**parallèle**".

Tampon (Buffer)

Les flux du NetFramework sont tous par défauts "bufférisés" contrairement à Java. Dans le NetFramework, un flux (un objet de classe Stream) est une abstraction d'une séquence d'octets, telle qu'un fichier, un périphérique d'entrée/sortie, un canal de communication à processus interne, ou un socket TCP/IP. En C#, un objet flux de classe Stream possède une propriété de longueur **Length** qui indique combien d'octets peuvent être traités par le flux. En outre un objet flux possède aussi une méthode **SetLength(long val)** qui définit la longueur du flux : cette mémoire intermédiaire associée au flux est aussi appelée **mémoire tampon** (**Buffer** en Anglais) du flux.

Lorsqu'un flux travaille sur des caractères, on peut faire que l'application lise ou écrive les caractères les uns après les autres en réglant la longueur du flux à 1 caractère (correspondance avec les flux non bufférisés de Java) :



On peut aussi faire que l'application lise ou écrive les caractères par groupes en réglant la longueur du flux à n caractères (correspondance avec les flux bufférisés de Java) :



2.3 Création d'un fichier de texte avec des données

Exemple d'utilisation des 2 classes précédentes.

Supposons que nous ayons l'architecture suivante sur le disque C:



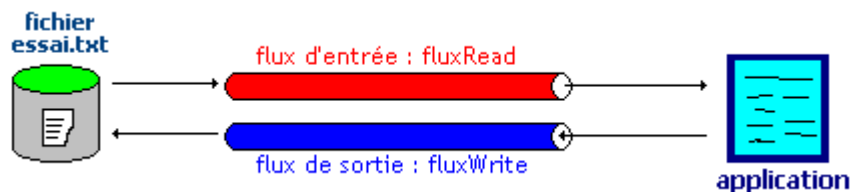
Il est possible de créer un nouveau fichier sur le disque dur et d'ouvrir un flux en écriture sur ce fichier en utilisant le constructeur de la classe **System.IO.StreamWriter** :

```
StreamWriter fluxWrite = new StreamWriter(@"c:\temp\rep1\essai.txt");
```

Voici le résultat obtenu par la ligne de code précédente sur le disque dur :



Le programme C# ci-dessous permet de créer le fichier texte nommé **essai.txt** et d'écrire des lignes de texte dans ce fichier avec la méthode **WriteLine(...)** de la classe **StreamWriter**, puis de lire le contenu selon le schéma ci-après avec la méthode **ReadLine()** de la classe **StreamReader** :



Code source C# correspondant :

```
if (!File.Exists( @"c:\temp\rep1\essai.txt" )) {
    // création d'un fichier texte et ouverture d'un flux en écriture sur ce fichier
    StreamWriter fluxWrite = new StreamWriter(@"c:\temp\rep1\essai.txt");
    Console.WriteLine("Fichier essai.text créé sur le disque dur.");
}
```

```

Console.WriteLine("Il n'écrase pas les données déjà présentes");

// écriture de lignes de texte dans le fichier à travers le flux :
for ( int i = 1; i<10; i++)
    fluxWrite.WriteLine("texte stocké par programme ligne N : "+i);
    fluxWrite.Close(); // fermeture du flux impérative pour sauvegarde des données
}

Console.WriteLine("Contenu du fichier essai.txt déjà présent :");
// création et ouverture d'un flux en lecture sur ce fichier
StreamReader fluxRead = new StreamReader(@"c:\temp\rep1\essai.txt");

// lecture des lignes de texte dans le fichier à travers le flux
string ligne;
while ( ( ligne = fluxRead.ReadLine()) != null )
    Console.WriteLine(ligne);
fluxRead.Close(); // fermeture du flux

```

2.4 Copie des données d'un fichier texte dans un autre

Nous utilisons l'instruction **using** qui définit un bloc permettant d'instancier un objet local au bloc à la fin duquel un objet est désalloué.

Un bloc **using** instanciant un objet de flux en lecture :

```
using ( StreamReader fluxRead = new StreamReader( ...)) { ....BLOC.... }
```

Un bloc **using** instanciant un objet de flux en écriture :

```
using (StreamWriter fluxWrite = new StreamWriter( ...)) { ....BLOC.... }
```

Code source C# créant le fichier essai.txt :

```

if ( !File.Exists( @"c:\temp\rep1\essai.txt" )) {

    using (StreamWriter fluxWrite = new StreamWriter(@"c:\temp\rep1\essai.txt"))
    {
        Console.WriteLine("Fichier essai.text créé sur le disque dur.");
        Console.WriteLine("Il n'écrase pas les données déjà présentes");
        // écriture de lignes de texte dans le fichier à travers le flux :
        for ( int i = 1; i<10; i++)
            fluxWrite.WriteLine("texte stocké par programme ligne N : "+i);
        } // fermeture et désallocation de l'objet fluxWrite
    }
}

```

Code source C# lisant le contenu du fichier essai.txt :

```

Console.WriteLine("Contenu du fichier 'essai.text' déjà présent :");
using ( StreamReader fluxRead = new StreamReader(@"c:\temp\rep1\essai.txt"))
{
    string ligne;

```

```

// lecture des lignes de texte dans le fichier à travers le flux :
while ((ligne = fluxRead.ReadLine()) != null )
    Console.WriteLine(ligne);
    Console.WriteLine("\nRecopie en cours ...");
} // fermeture et désallocation de l'objet fluxRead

```

Code source C# recopiant le contenu du fichier `essai.txt`, créant le fichier `CopyEssai.txt` et recopiant le contenu de `essai.txt` :

```

using ( StreamReader fluxRead = new StreamReader(@"c:\temp\rep1\essai.txt"))
{
    StreamWriter fluxWrite = new StreamWriter(@"c:\temp\rep1\CopyEssai.txt");
    string ligne;
    // on lit dans essai.txt à travers fluxRead et on écrit dans
    // CopyEssai.txt à travers fluxWrite :
    while ((ligne = fluxRead.ReadLine()) != null )
        fluxWrite.WriteLine("copie < "+ligne+" >");
    fluxWrite.Close();// fermeture de fluxWrite
} // fermeture et désallocation de l'objet fluxRead

```

Code source C# lisant le contenu du fichier `CopyEssai.txt` :

```

using ( StreamReader fluxRead = new StreamReader(@"c:\temp\rep1\essai.txt"))
{
    Console.WriteLine("\nContenu de la copie 'copyEssai.txt' :");
    string ligne;
    // lecture des lignes de texte du nouveau fichier copié fichier à travers le flux :
    while ((ligne = fluxRead.ReadLine()) != null )
        Console.WriteLine(ligne);
} // fermeture et désallocation de l'objet fluxRead

```

Résultat d'exécution du code précédent :

Contenu du fichier '`essai.txt`' déjà présent :

texte stocké par programme ligne N : 1

texte stocké par programme ligne N : 2

.....

texte stocké par programme ligne N : 9

Recopie en cours ...

Contenu de la copie '`copyEssai.txt`' :

copie < texte stocké par programme ligne N : 1 >

copie < texte stocké par programme ligne N : 2 >

.....

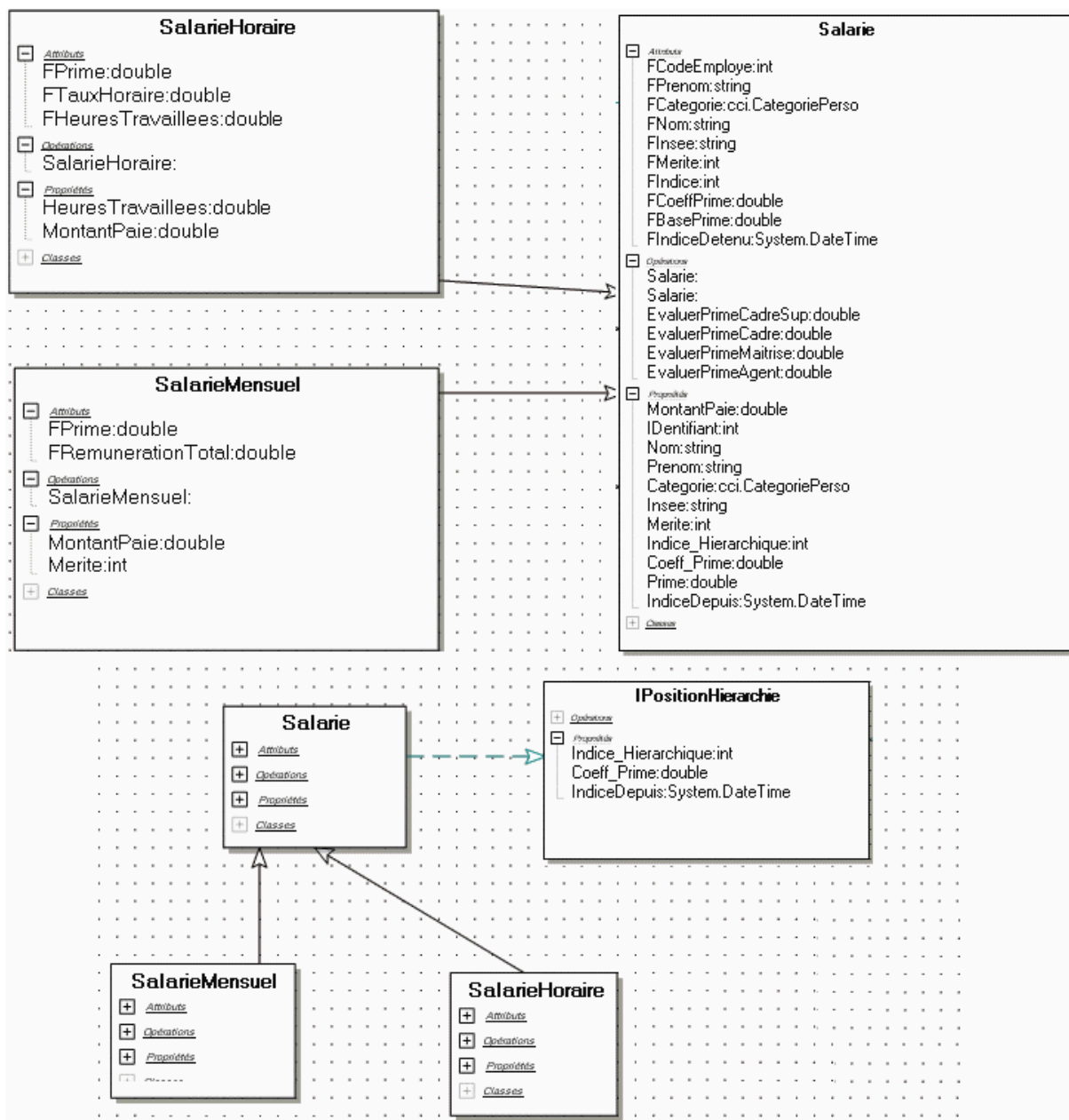
copie < texte stocké par programme ligne N : 9 >

3. Exercice de fichier de salariés avec une IHM

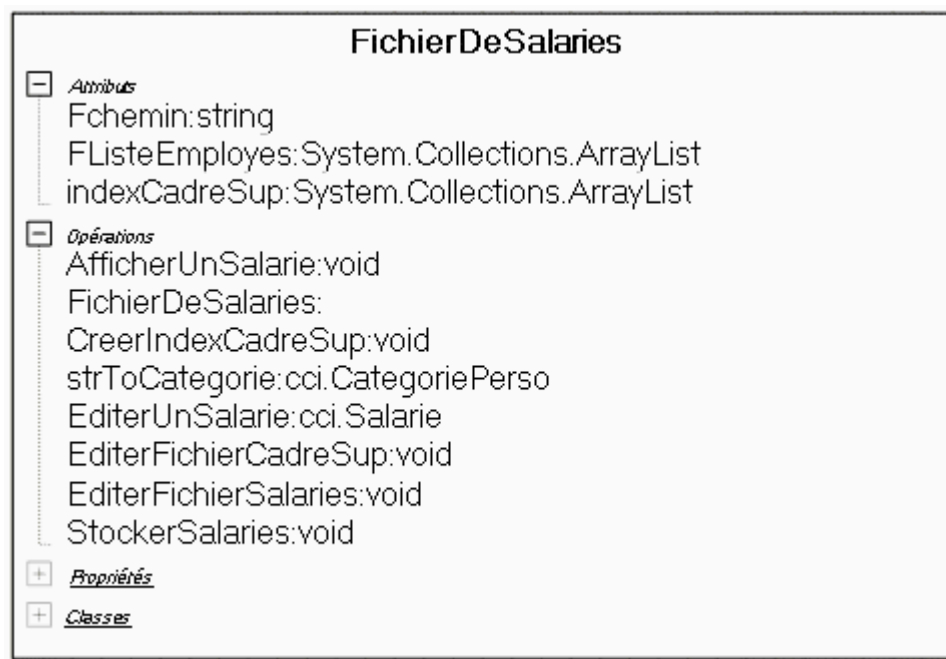
Soit à écrire un programme C# gérant en saisie et en consultation un fichier des salariés d'une petite entreprise.

Ci-dessous les éléments à écrire en mode console d'abord, puis ensuite créez vous-mêmes une interface interactive.

Soit les diagrammes de classe suivants :



Le programme travaille sur un fichier d'objets de classe Salarie :



Un exemple d'IHM possible pour ce programme de gestion de salariés :

World Company - consultation sur un salarié du fichier

nom coef. prime annuelle
 prénom
 INSEE
 Mérite: ☐ 1 ☐ 4 ☐ 2 ☐ 5 ☐ 3 ☒ 6
 Catégorie: ☒ Cadre_Sup ☐ Agent ☐ Cadre ☐ Autre ☐ Maîtrise
 indice

dimanche 14 août 2005
 123456 : Euton, Jeanne
 123457 : Yonaize, Mah
 123458 : Ziaire, Marie
 123459 : Louga, Belle
 123460 : Miett, Hamas
 123461 : Kong, King
 123462 : Zaume, Philippo
 123463 : Micoton, Mylène
 123464 : Labbé, terave
 123465 : Vernes, jules
 123466 : Milou, canis
 123467 : Richelieu, francoise
 123468 : Nightingale, Florence
 123469 : Emerson, Ralph
 123470 : Jaurès, Jean
 123471 : Lajoie, Philomène
 123472 : Abrial, Noémie
 123473 : Amurabi, net
 123474 : Lemon, Janette
 123475 : Paré, Ambroise
 123476 : Heine, Henri

montant prime annuelle
 salaire mensuel effectif

Enregistrer la saisie

Saisie **Consultation**

Les informations afférentes à un salarié de l'entreprise sont stockées dans un fichier **Fiches** qui est un objet de classe FichierDeSalaries qui se trouve stocké sur le disque dur sous le nom de fichierSalaries.txt. Le programme travaille en mémoire centrale sur une image de ce fichier qui est rangée dans un **ArrayList** nommé ListeSalaries :

```
ArrayList ListeSalaries = new ArrayList ( ) ;
```

```
FichierDeSalaries Fiches = new FichierDeSalaries ("fichierSalaries.txt" , ListeSalaries );
```

Squelette et implémentation partielle proposés des classes de base :

```
enum CategoriePerso
{
    Cadre_Sup,Cadre,Maitrise,Agent,Autre
}

/// <summary>
/// Interface définissant les propriétés de position d'un
/// salarié dans la hiérarchie de l'entreprise.
/// </summary>
interface IPositionHierarchie {
    int Indice_Hierarchique
    {
        get ;
        set ;
    }
    double Coeff_Prime
    {
        get ;
        set ;
    }
    DateTime IndiceDepuis
    {
        get ;
        set ;
    }
} // fin interface IPositionHierarchie
/// <summary>
/// Classe de base abstraite pour le personnel. Cette classe n'est
/// pas instanciable.
/// </summary>
abstract class Salarie : IPositionHierarchie {
    /// attributs identifiant le salarié :
    private int FCodeEmploye ;
    private string FPrenom ;
    private CategoriePerso FCategorye ;
    private string FNom ;
    private string FInsee ;
    protected int FMerite ;
```

```

private int FIndice ;
private double FCoeffPrime ;
private double FBasePrime ;
private DateTime FIndiceDetenu ;

///le constructeur de la classe employé au mérite :
public Salarie ( int IDentifiant, string Nom, string Prenom, CategoriePerso Categorie,
string Insee, int Merite, int Indice, double CoeffPrime )
{
    FCodeEmploye = IDentifiant ;
    FNom = Nom ;
    FPrenom = Prenom ;
    FCategorie = Categorie ;
    FInsee = Insee ;
    FMerite = Merite ;
    FIndice = Indice ;
    FCoeffPrime = CoeffPrime ;
    FIndiceDetenu = DateTime.Now ;
    switch ( FCategorie ) {
        case CategoriePerso.Cadre_Sup : FBasePrime = 2000 ; break;
        case CategoriePerso.Cadre : FBasePrime = 1000 ; break;
        case CategoriePerso.Maitrise : FBasePrime = 500 ; break;
        case CategoriePerso.Agent : FBasePrime = 200 ; break;
    }
}

///le constructeur de la classe employé sans mérite :
public Salarie ( int IDentifiant, string Nom, string Prenom, CategoriePerso Categorie, string
Insee ) : this( IDentifiant, Nom, Prenom, Categorie, Insee,0,0,0 ) {

}

protected double EvaluerPrimeCadreSup ( int coeffMerite ) {
    return ( 100 + coeffMerite * 8 ) * FCoeffPrime * FBasePrime + FIndice * 7 ;
}
protected double EvaluerPrimeCadre ( int coeffMerite ) {
    return ( 100 + coeffMerite * 6 ) * FCoeffPrime * FBasePrime + FIndice * 5 ;
}
protected double EvaluerPrimeMaitrise ( int coeffMerite ) {
    return ( 100 + coeffMerite * 4 ) * FCoeffPrime * FBasePrime + FIndice * 3 ;
}
protected double EvaluerPrimeAgent ( int coeffMerite ) {
    return ( 100 + coeffMerite * 2 ) * FCoeffPrime * FBasePrime + FIndice * 2 ;
}

/// propriété abstraite donnant le montant du salaire
/// (virtual automatiquement)
abstract public double MontantPaie { get ; }
/// propriété identifiant le salarié dans l'entreprise :
public int IDentifiant {
    get { return FCodeEmploye ; }
}

/// propriété nom du salarié :

```

```

public string Nom {
    get { return FNom; }
    set { FNom = value; }
}

/// propriété nom du salarié :
public string Prenom {
    get { return FPrenom; }
    set { FPrenom = value; }
}

/// propriété catégorie de personnel du salarié :
public CatégoriePerso Catégorie {
    get { return FCatégorie; }
    set { FCatégorie = value; }
}

/// propriété n° de sécurité sociale du salarié :
public string Insee {
    get { return FInsee; }
    set { FInsee = value; }
}

/// propriété de point de mérite du salarié :
public virtual int Merite {
    get { return FMerite; }
    set { FMerite = value; }
}

/// propriété classement indiciaire dans la hiérarchie :
public int Indice_Hierarchique {
    get { return FIndice; }
    set {
        FIndice = value;
        //--Maj de la date de détention du nouvel indice :
        IndiceDepuis = DateTime.Now;
    }
}

/// propriété coefficient de la prime en %:
public double Coeff_Prime {
    get { return FCoeffPrime; }
    set { FCoeffPrime = value; }
}

/// propriété valeur de la prime :
public double Prime {
    get {
        switch (FCatégorie)
        {
            case CatégoriePerso.Cadre_Sup : return EvaluerPrimeCadreSup ( FMerite );
            case CatégoriePerso.Cadre : return EvaluerPrimeCadre ( FMerite );
            case CatégoriePerso.Maitrise : return EvaluerPrimeMaitrise ( FMerite );
            case CatégoriePerso.Agent : return EvaluerPrimeAgent ( FMerite );
            default : return EvaluerPrimeAgent ( 0 );
        }
    }
}

```

```

    /// date à laquelle l'indice actuel a été obtenu :
    public DateTime IndiceDepuis {
        get { return FIndiceDetenu ; }
        set { FIndiceDetenu = value ; }
    }
} // fin classe Salarie

/// <summary>
/// Classe du personnel mensualisé. Implémente la propriété abstraite
/// MontantPaie déclarée dans la classe de base (mère).
/// </summary>
class SalarieMensuel : Salarie
{
    /// attributs du salaire annuel :
    private double FPrime ;
    private double FRemunerationTotal ;

    ///le constructeur de la classe (salarié au mérite) :
    public SalarieMensuel ( int IDentifiant, string Nom, string Prenom, CategoriePerso
    Categorie, string Insee, int Merite, int Indice, double CoeffPrime, double
    RemunerationTotal ) :base ( IDentifiant, Nom, Prenom, Categorie, Insee, Merite, Indice,
    CoeffPrime )
    {
        FPrime = this.Prime ;
        FRemunerationTotal = RemunerationTotal ;
    }

    /// implémentation de la propriété donnant le montant du salaire :
    public override double MontantPaie {
        get { return ( FRemunerationTotal + this.Prime ) / 12 ; }
    }

    /// propriété de point de mérite du salarié :
    public override int Merite {
        get { return FMerite ; }
        set {
            FMerite = value ;
            FPrime = this.Prime ;
        }
    }
} // fin classe SalarieMensuel

/// <summary>
/// Classe du personnel horaire. Implemente la propriété abstraite
/// MontantPaie déclarée dans la classe de base (mère).
/// </summary>
class SalarieHoraire : Salarie
{
    /// attributs permettant le calcul du salaire :
    private double FPrime ;
    private double FTauxHoraire ;

```

```

private double FHeuresTravaillees ;

//le constructeur de la classe (salarié non au mérite):
public SalarieHoraire ( int IDentifiaint, string Nom, string Prenom, string Insee, double
TauxHoraire ): base ( IDentifiaint, Nom, Prenom, CategoriePerso.Autre, Insee )
{
    FTauxHoraire = TauxHoraire ;
    FHeuresTravaillees = 0 ;
    FPrime = 0 ;
}
// nombre d'heures effectuées :
public double HeuresTravaillees {
    get { return FHeuresTravaillees ; }
    set { FHeuresTravaillees = value ; }
}
// implémentation de la propriété donnant le montant du salaire :
public override double MontantPaie {
    get { return FHeuresTravaillees * FTauxHoraire + FPrime ; }
}
} // fin classe SalarieHoraire

class FichierDeSalaries
{
    private string Fchemin ;
    private ArrayList FListeEmployes ; // liste des nouveaux employés à entrer dans le fichier
    private ArrayList indexCadreSup ; // Table d'index des cadres supérieurs du fichier

    // méthode static affichant un objet Salarie à la console :
    public static void AfficherUnSalarie ( Salarie Employe ) {
        // pour l'instant un salarié mensualisé seulement
    }

    // constructeur de la classeFichierDeSalaries
    public FichierDeSalaries ( string chemin, ArrayList Liste ) {

    }

    // méthode de création de la table d'index des cadre_sup :
    public void CreerIndexCadreSup () {

    }

    // méthode convertissant le champ string catégorie en la constante enum associée
    private CategoriePerso strToCategorie ( string s ) {

    }

    // méthode renvoyant un objet SalarieMensuel de rang fixé dans le fichier
    private Salarie EderUnSalarie ( int rang ) {
        SalarieMensuel perso ;

        .....
        perso = new SalarieMensuel ( IDentifiaint, Nom, Prenom, Categorie, Insee,
            Merite, Indice, CoeffPrime, RemunerationTotal );
        .....
        return perso ;
    }
}

```

```

}
// méthode affichant sur la console à partir de la table d'index :
public void EditerFichierCadreSup ( )
{
    .....
    foreach( int ind in indexCadreSup )
    {
        AfficherUnSalarie ( EditerUnSalarie ( ind ) );
    }
    .....
}
// méthode affichant sur la console le fichier de tous les salariés :
public void EditerFichierSalaries ( ) {

}
// méthode créant et stockant des salariés dans le fichier :
public void StockerSalaries ( ArrayList ListeEmploy )
{
    .....
    // si le fichier n'existe pas => création du fichier sur disque :
    StreamWriter fichierSortie = File.CreateText ( Fchemin );
    fichierSortie.WriteLine ("Fichier des personnels");
    fichierSortie.Close ();
    .....
    // ajout dans le fichier de toute la liste :
    .....
    foreach( Salarie s in ListeEmploy )
    {

    }
    .....
}
} // fin classe FichierDeSalaries

```

Implémenter les classes avec le programme de test suivant :

```

class ClassUsesSalarie
{
    /// <summary>
    /// Le point d'entrée principal de l'application.
    /// </summary>
    static void InfoSalarie ( SalarieMensuel empl )
    {
        FichierDeSalaries.AfficherUnSalarie ( empl );
        double coefPrimeLoc = empl.Coeff_Prime ;
        int coefMeriteLoc = empl.Merite ;
        //--impact variation du coef de prime
        for( double i = 0.5 ; i < 1 ; i += 0.1 )
        {
            empl.Coeff_Prime = i ;
        }
    }
}

```

```

    Console.WriteLine ( "  coeff prime : " + empl.Coeff_Prime );
    Console.WriteLine ( "  montant prime annuelle : " + empl.Prime );
    Console.WriteLine ( "  montant paie mensuelle: " + empl.MontantPaie );
}
Console.WriteLine ( "  -----");
empl.Coeff_Prime = coefPrimeLoc ;
//--impact variation du coef de mérite
for( int i = 0 ; i < 10 ; i ++ )
{
    empl.Merite = i ;
    Console.WriteLine ( "  coeff mérite : " + empl.Merite );
    Console.WriteLine ( "  montant prime annuelle : " + empl.Prime );
    Console.WriteLine ( "  montant paie mensuelle: " + empl.MontantPaie );
}
empl.Merite = coefMeriteLoc ;
Console.WriteLine ("=====");

}
[STAThread]
static void Main ( string [] args )
{
    SalarieMensuel Employee1 = new SalarieMensuel ( 123456, "Euton" , "Jeanne" ,
        CategoriePerso.Cadre_Sup, "2780258123456" ,6,700,0.5,50000 );
    SalarieMensuel Employee2 = new SalarieMensuel ( 123457, "Yonaize" , "Mah" ,
        CategoriePerso.Cadre, "1821113896452" ,5,520,0.42,30000 );
    SalarieMensuel Employee3 = new SalarieMensuel ( 123458, "Ziaire" , "Marie" ,
        CategoriePerso.Maitrise, "2801037853781" ,2,678,0.6,20000 );
    SalarieMensuel Employee4 = new SalarieMensuel ( 123459, "Louga" , "Belle" ,
        CategoriePerso.Agent, "2790469483167" ,4,805,0.25,20000 );

    ArrayList ListeSalaries = new ArrayList ();
    ListeSalaries.Add ( Employee1 );
    ListeSalaries.Add ( Employee2 );
    ListeSalaries.Add ( Employee3 );
    ListeSalaries.Add ( Employee4 );
    foreach( SalarieMensuel s in ListeSalaries )
        InfoSalarie ( s );
    Console.WriteLine (">>> Promotion indice de " + Employee1.Nom + " dans 2 secondes.");
    Thread.Sleep ( 2000 );
    Employee1.Indice_Hierarchique = 710 ;
    InfoSalarie ( Employee1 );
    //-----//
    FichierDeSalaries Fiches = new FichierDeSalaries ("fichierSalaries.txt" ,ListeSalaries );
    Console.WriteLine (">>> Attente 3 s pour création de nouveaux salariés");
    Thread.Sleep ( 3000 );
    Employee1 = new SalarieMensuel ( 123460, "Miett" , "Hamas" ,
        CategoriePerso.Cadre_Sup, "1750258123456" ,4,500,0.7,42000 );
    Employee2 = new SalarieMensuel ( 123461, "Kong" , "King" ,
        CategoriePerso.Cadre, "1640517896452" ,4,305,0.62,28000 );
    Employee3 = new SalarieMensuel ( 123462, "Zaume" , "Philippo" ,
        CategoriePerso.Maitrise, "1580237853781" ,2,245,0.8,15000 );

```

```

    Employe4 = new SalarieMensuel ( 123463, "Micoton" , "Mylène" ,
        CategoriePerso.Agent, "2850263483167" ,4,105,0.14,12000 );
    ListeSalaries = new ArrayList ();
    ListeSalaries.Add ( Employe1 );
    ListeSalaries.Add ( Employe2 );
    ListeSalaries.Add ( Employe3 );
    ListeSalaries.Add ( Employe4 );
    Fiches.StockerSalaries ( ListeSalaries );
    Fiches.EditerFichierSalaries ();
    Fiches.CreerIndexCadreSup ();
    Fiches.EditerFichierCadreSup ();
    System.Console.ReadLine ();
}
}
}

```

Exemple de résultats obtenus avec le programme de test précédent :

fichierSalaries.txt :

```

Fichier des personnels
123456
Euton
Jeanne
*Cadre_Sup
2780258123456
6
710
15/02/2004 19:52:38
0,5
152970
16914,16666666667
123457
Yonaize
Mah
*Cadre
1821113896452
5
520
15/02/2004 19:52:36
0,42
57200
7266,66666666667
123458
Ziaire
Marie
*Maitrise
2801037853781
2
678
15/02/2004 19:52:36

```

0,6
34434
4536,16666666667
123459
Loug
Belle
*Agent
2790469483167
4
805
15/02/2004 19:52:36
0,25
7010
2250,83333333333
123460
Miett
Ham
*Cadre_Sup
1750258123456
4
500
15/02/2004 19:52:41
0,7
188300
19191,66666666667
123461
Kong
King
*Cadre
1640517896452
4
305
15/02/2004 19:52:41
0,62
78405
8867,08333333333
123462
Zaume
Philippo
*Maitrise
1580237853781
2
245
15/02/2004 19:52:41
0,8
43935
4911,25
123463
Micoton
Mylène
*Agent

2850263483167
4
105
15/02/2004 19:52:41
0,14
3234
1269,5

Résultats console :

Employé n°123456: Euton / Jeanne
n° SS : 2780258123456
catégorie : Cadre_Sup
indice hiérarchique : 700 , détenu depuis : 15/02/2004 19:52:36
coeff mérite : 6
coeff prime : 0,5
montant prime annuelle : 152900
montant paie mensuelle: 16908,3333333333
coeff prime : 0,5
montant prime annuelle : 152900
montant paie mensuelle: 16908,3333333333
coeff prime : 0,6
montant prime annuelle : 182500
montant paie mensuelle: 19375
coeff prime : 0,7
montant prime annuelle : 212100
montant paie mensuelle: 21841,6666666667
coeff prime : 0,8
montant prime annuelle : 241700
montant paie mensuelle: 24308,3333333333
coeff prime : 0,9
montant prime annuelle : 271300
montant paie mensuelle: 26775
coeff prime : 1
montant prime annuelle : 300900
montant paie mensuelle: 29241,6666666667

coeff mérite : 0
montant prime annuelle : 104900
montant paie mensuelle: 12908,3333333333
coeff mérite : 1
montant prime annuelle : 112900
montant paie mensuelle: 13575
coeff mérite : 2
montant prime annuelle : 120900
montant paie mensuelle: 14241,6666666667
coeff mérite : 3
montant prime annuelle : 128900
montant paie mensuelle: 14908,3333333333
coeff mérite : 4
montant prime annuelle : 136900

montant paie mensuelle: 15575
coeff mérite : 5
montant prime annuelle : 144900
montant paie mensuelle: 16241,6666666667
coeff mérite : 6
montant prime annuelle : 152900
montant paie mensuelle: 16908,3333333333
coeff mérite : 7
montant prime annuelle : 160900
montant paie mensuelle: 17575
coeff mérite : 8
montant prime annuelle : 168900
montant paie mensuelle: 18241,6666666667
coeff mérite : 9
montant prime annuelle : 176900
montant paie mensuelle: 18908,3333333333

=====

Employé n°123457: Yonaize / Mah
SS : 1821113896452
catégorie : Cadre
indice hiérarchique : 520 , détenu depuis : 15/02/2004 19:52:36
coeff mérite : 5
coeff prime : 0,42
montant prime annuelle : 57200
montant paie mensuelle: 7266,6666666667
coeff prime : 0,5
montant prime annuelle : 67600
montant paie mensuelle: 8133,3333333333
coeff prime : 0,6
montant prime annuelle : 80600
montant paie mensuelle: 9216,6666666667
coeff prime : 0,7
montant prime annuelle : 93600
montant paie mensuelle: 10300
coeff prime : 0,8
montant prime annuelle : 106600
montant paie mensuelle: 11383,3333333333
coeff prime : 0,9
montant prime annuelle : 119600
montant paie mensuelle: 12466,6666666667
coeff prime : 1
montant prime annuelle : 132600
montant paie mensuelle: 13550

coeff mérite : 0
montant prime annuelle : 44600
montant paie mensuelle: 6216,6666666667
coeff mérite : 1
montant prime annuelle : 47120
montant paie mensuelle: 6426,6666666667
coeff mérite : 2

montant prime annuelle : 49640
 montant paie mensuelle: 6636,66666666667
 coeff mérite : 3
 montant prime annuelle : 52160
 montant paie mensuelle: 6846,66666666667
 coeff mérite : 4
 montant prime annuelle : 54680
 montant paie mensuelle: 7056,66666666667
 coeff mérite : 5
 montant prime annuelle : 57200
 montant paie mensuelle: 7266,66666666667
 coeff mérite : 6
 montant prime annuelle : 59720
 montant paie mensuelle: 7476,66666666667
 coeff mérite : 7
 montant prime annuelle : 62240
 montant paie mensuelle: 7686,66666666667
 coeff mérite : 8
 montant prime annuelle : 64760
 montant paie mensuelle: 7896,66666666667
 coeff mérite : 9
 montant prime annuelle : 67280
 montant paie mensuelle: 8106,66666666667

=====

Employé n°123458: Ziaire / Marie
 n°SS : 2801037853781
 catégorie : Maitrise
 indice hiérarchique : 678 , détenu depuis : 15/02/2004 19:52:36
 coeff mérite : 2
 coeff prime : 0,6
 montant prime annuelle : 34434
 montant paie mensuelle: 4536,16666666667
 coeff prime : 0,5
 montant prime annuelle : 29034
 montant paie mensuelle: 4086,16666666667
 coeff prime : 0,6
 montant prime annuelle : 34434
 montant paie mensuelle: 4536,16666666667
 coeff prime : 0,7
 montant prime annuelle : 39834
 montant paie mensuelle: 4986,16666666667
 coeff prime : 0,8
 montant prime annuelle : 45234
 montant paie mensuelle: 5436,16666666667
 coeff prime : 0,9
 montant prime annuelle : 50634
 montant paie mensuelle: 5886,16666666667
 coeff prime : 1
 montant prime annuelle : 56034
 montant paie mensuelle: 6336,16666666667

coeff mérite : 0
 montant prime annuelle : 32034
 montant paie mensuelle: 4336,16666666667
 coeff mérite : 1
 montant prime annuelle : 33234
 montant paie mensuelle: 4436,16666666667
 coeff mérite : 2
 montant prime annuelle : 34434
 montant paie mensuelle: 4536,16666666667
 coeff mérite : 3
 montant prime annuelle : 35634
 montant paie mensuelle: 4636,16666666667
 coeff mérite : 4
 montant prime annuelle : 36834
 montant paie mensuelle: 4736,16666666667
 coeff mérite : 5
 montant prime annuelle : 38034
 montant paie mensuelle: 4836,16666666667
 coeff mérite : 6
 montant prime annuelle : 39234
 montant paie mensuelle: 4936,16666666667
 coeff mérite : 7
 montant prime annuelle : 40434
 montant paie mensuelle: 5036,16666666667
 coeff mérite : 8
 montant prime annuelle : 41634
 montant paie mensuelle: 5136,16666666667
 coeff mérite : 9
 montant prime annuelle : 42834
 montant paie mensuelle: 5236,16666666667

=====

Employé n°123459: Louga / Belle
 n°SS : 2790469483167
 catégorie : Agent
 indice hiérarchique : 805 , détenu depuis : 15/02/2004 19:52:36
 coeff mérite : 4
 coeff prime : 0,25
 montant prime annuelle : 7010
 montant paie mensuelle: 2250,83333333333
 coeff prime : 0,5
 montant prime annuelle : 12410
 montant paie mensuelle: 2700,83333333333
 coeff prime : 0,6
 montant prime annuelle : 14570
 montant paie mensuelle: 2880,83333333333
 coeff prime : 0,7
 montant prime annuelle : 16730
 montant paie mensuelle: 3060,83333333333
 coeff prime : 0,8
 montant prime annuelle : 18890
 montant paie mensuelle: 3240,83333333333

coeff prime : 0,9
montant prime annuelle : 21050
montant paie mensuelle: 3420,833333333333
coeff prime : 1
montant prime annuelle : 23210
montant paie mensuelle: 3600,833333333333

coeff mérite : 0
montant prime annuelle : 6610
montant paie mensuelle: 2217,5
coeff mérite : 1
montant prime annuelle : 6710
montant paie mensuelle: 2225,833333333333
coeff mérite : 2
montant prime annuelle : 6810
montant paie mensuelle: 2234,166666666667
coeff mérite : 3
montant prime annuelle : 6910
montant paie mensuelle: 2242,5
coeff mérite : 4
montant prime annuelle : 7010
montant paie mensuelle: 2250,833333333333
coeff mérite : 5
montant prime annuelle : 7110
montant paie mensuelle: 2259,166666666667
coeff mérite : 6
montant prime annuelle : 7210
montant paie mensuelle: 2267,5
coeff mérite : 7
montant prime annuelle : 7310
montant paie mensuelle: 2275,833333333333
coeff mérite : 8
montant prime annuelle : 7410
montant paie mensuelle: 2284,166666666667
coeff mérite : 9
montant prime annuelle : 7510
montant paie mensuelle: 2292,5

=====
>>> Promotion indice de Euton dans 2 secondes.
Employé n°123456: Euton / Jeanne
n° SS : 2780258123456
catégorie : Cadre_Sup
indice hiérarchique : 710 , détenu depuis : 15/02/2004 19:52:38
coeff mérite : 6
coeff prime : 0,5
montant prime annuelle : 152970
montant paie mensuelle: 16914,166666666667
coeff prime : 0,5
montant prime annuelle : 152970
montant paie mensuelle: 16914,166666666667
coeff prime : 0,6

montant prime annuelle : 182570
montant paie mensuelle: 19380,8333333333
coeff prime : 0,7
montant prime annuelle : 212170
montant paie mensuelle: 21847,5
coeff prime : 0,8
montant prime annuelle : 241770
montant paie mensuelle: 24314,1666666667
coeff prime : 0,9
montant prime annuelle : 271370
montant paie mensuelle: 26780,8333333333
coeff prime : 1
montant prime annuelle : 300970
montant paie mensuelle: 29247,5

coeff mérite : 0
montant prime annuelle : 104970
montant paie mensuelle: 12914,1666666667
coeff mérite : 1
montant prime annuelle : 112970
montant paie mensuelle: 13580,8333333333
coeff mérite : 2
montant prime annuelle : 120970
montant paie mensuelle: 14247,5
coeff mérite : 3
montant prime annuelle : 128970
montant paie mensuelle: 14914,1666666667
coeff mérite : 4
montant prime annuelle : 136970
montant paie mensuelle: 15580,8333333333
coeff mérite : 5
montant prime annuelle : 144970
montant paie mensuelle: 16247,5
coeff mérite : 6
montant prime annuelle : 152970
montant paie mensuelle: 16914,1666666667
coeff mérite : 7
montant prime annuelle : 160970
montant paie mensuelle: 17580,8333333333
coeff mérite : 8
montant prime annuelle : 168970
montant paie mensuelle: 18247,5
coeff mérite : 9
montant prime annuelle : 176970
montant paie mensuelle: 18914,1666666667

=====
>>> Attente 3 s pour création de nouveaux salariés
Fichier des personnels
123456
Euton
Jeanne

*Cadre_Sup
2780258123456
6
710
15/02/2004 19:52:38
0,5
152970
16914,1666666667
123457
Yonaize
Mah
*Cadre
1821113896452
5
520
15/02/2004 19:52:36
0,42
57200
7266,6666666667
123458
Ziaire
Marie
*Maitrise
2801037853781
2
678
15/02/2004 19:52:36
0,6
34434
4536,1666666667
123459
Louga
Belle
*Agent
2790469483167
4
805
15/02/2004 19:52:36
0,25
7010
2250,8333333333
123460
Miett
Hamas
*Cadre_Sup
1750258123456
4
500
15/02/2004 19:52:41
0,7
188300

19191,6666666667
 123461
 Kong
 King
 *Cadre
 1640517896452
 4
 305
 15/02/2004 19:52:41
 0,62
 78405
 8867,08333333333
 123462
 Zaume
 Philippo
 *Maitrise
 1580237853781
 2
 245
 15/02/2004 19:52:41
 0,8
 43935
 4911,25
 123463
 Micoton
 Mylène
 *Agent
 2850263483167
 4
 105
 15/02/2004 19:52:41
 0,14
 3234
 1269,5
 ++> *Cadre_Sup : 5
 ++> *Cadre_Sup : 49
 Employé n°123456: Euton / Jeanne
 n° : 2780258123456
 catégorie : Cadre_Sup
 indice hiérarchique : 710 , détenu depuis : 15/02/2004 19:52:38
 coeff mérite : 6
 coeff prime : 0
 montant prime annuelle : 4970
 montant paie mensuelle: 414,208333333333
 Employé n°123460: Mielt / Hamas
 n° : 1750258123456
 catégorie : Cadre_Sup
 indice hiérarchique : 500 , détenu depuis : 15/02/2004 19:52:41
 coeff mérite : 4
 coeff prime : 0
 montant prime annuelle : 3500

montant paie mensuelle: 291,725

Précisons que pour pouvoir utiliser l'instruction de simulation d'attente de 2 secondes entre deux promotions, soit `Thread.Sleep ( 2000 );` il est nécessaire de déclarer dans les clauses using :

`using System.Threading;`