Livret – 3.3

Le langage C# dans .Net

Propriétés, exceptions, flux, formulaires et contrôles.

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

SOMMAIRE

Propriétés et indexeurs en

Plan général:

1. Les propriétés

2. Les indexeurs

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

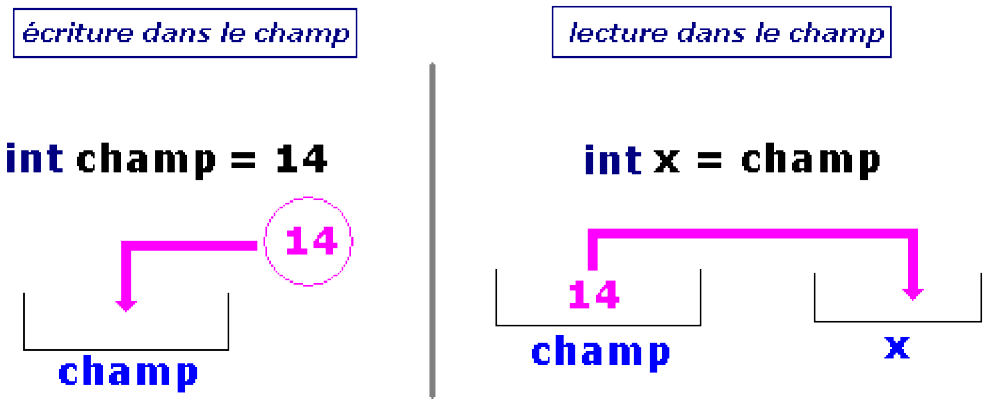
Définition d'une propriété

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 propr1 de type **int** :

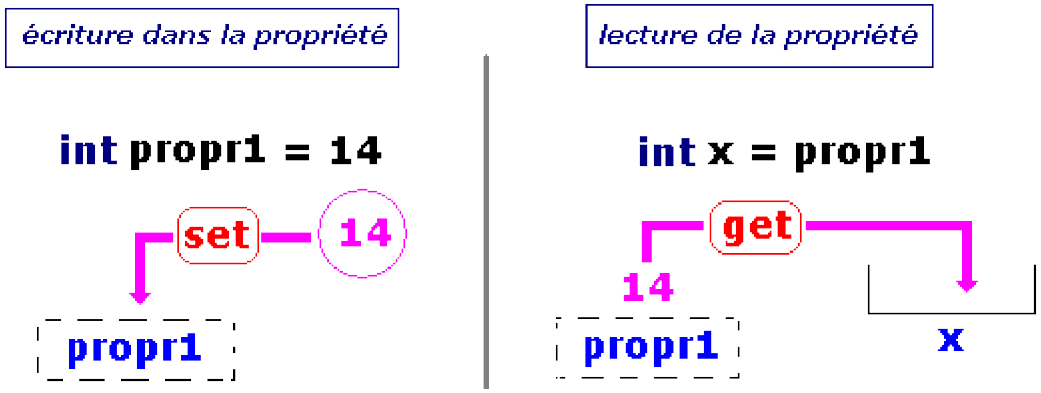
```
public int propr1 {
        //..... bloc de définition
}
```

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  ; }
}
```

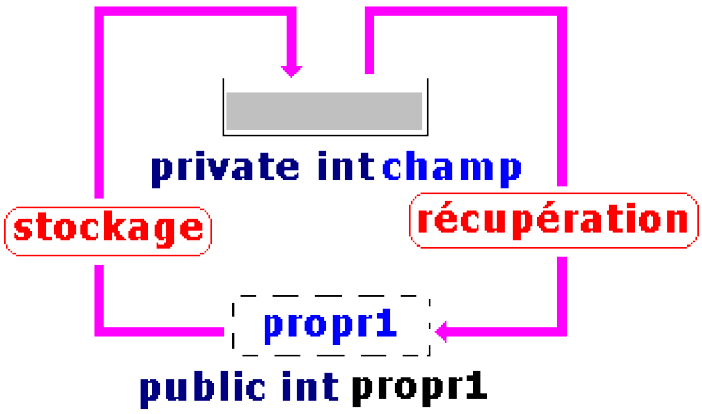
Le mot clef ***value*** est une sorte de paramètre implicite interne à l'accesseur **set**, il contient la valeur effective qui est transmise à la propriété lors de l'accès en écriture.

D'un 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 " :

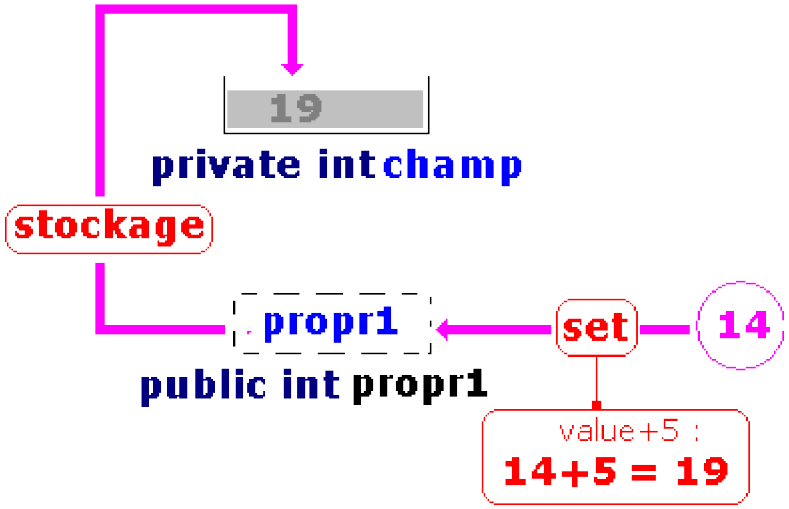
```
private int champ;

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

Utilisons cette propriété en mode écriture à travers une affectation :

```
prop1 = 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.*

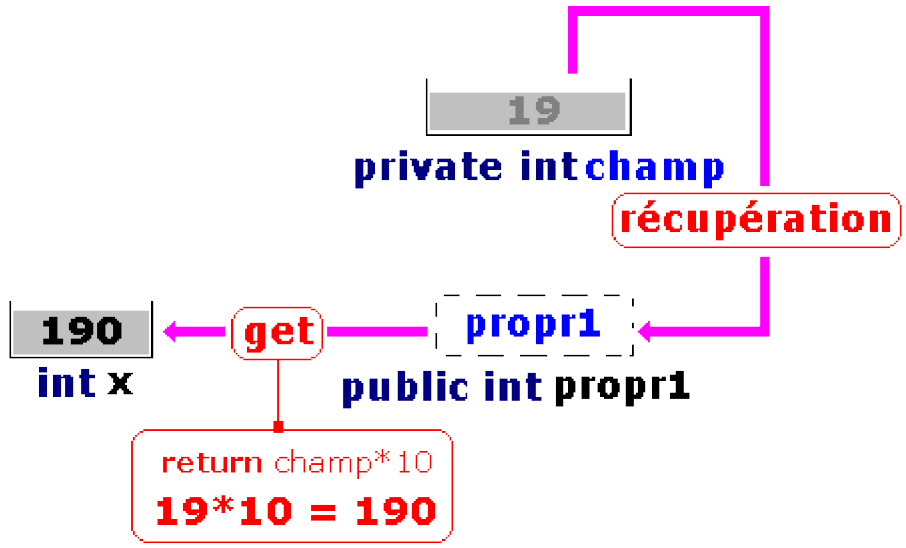


Utilisons maintenant notre propriété en mode lecture à travers une affectation :

```
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

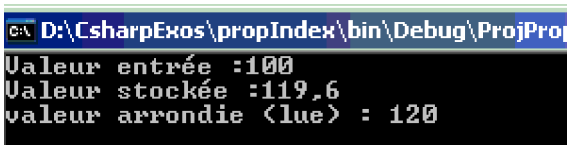
Une propriété servant à fournir automatiquement le prix d'un article en y intégrant la TVA au taux de 19.6% et arrondi à l'unité d'euro supérieur :

```
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 :



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*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 :

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 :

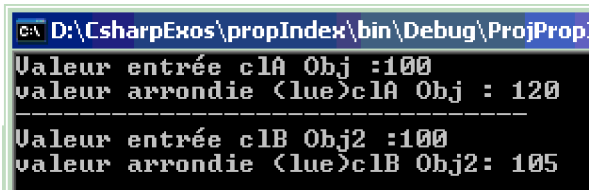
[C# Avertissement] Class..... : Le mot clé new est requis sur '.....', car il masque le membre hérité..... '

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 :



1.6 Les propriétés peuvent être virtuelles et redéfinies comme les méthodes

Les propriété 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.

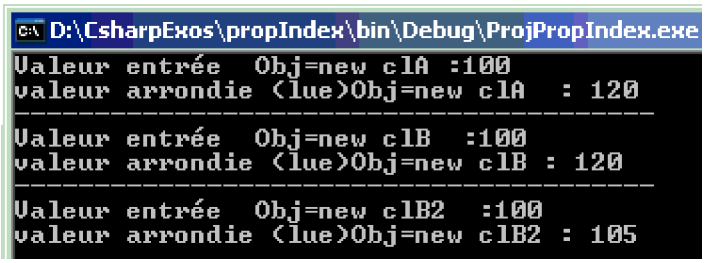
Notre objectif est de comparer les résultats d'exécution obtenus lorsque l'on utilise une référence d'objet de classe mère instanciée soit en objet de classe **clB** ou **clB2**. C'est le comportement de la propriété prix dans chacun de deux cas (statique ou

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 :



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 résulats 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 necéssiter 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 ; }
```

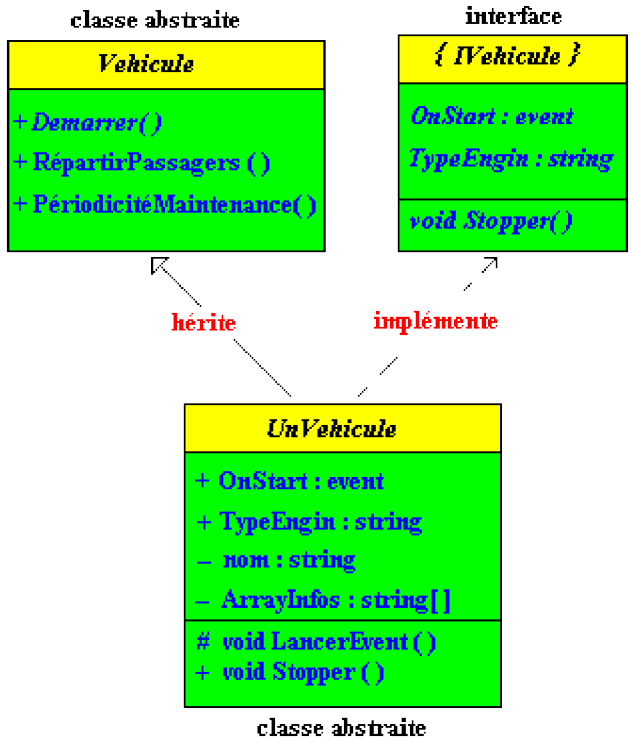
```
  }
 }

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+"]" ;      }
  }
  ....
}
```

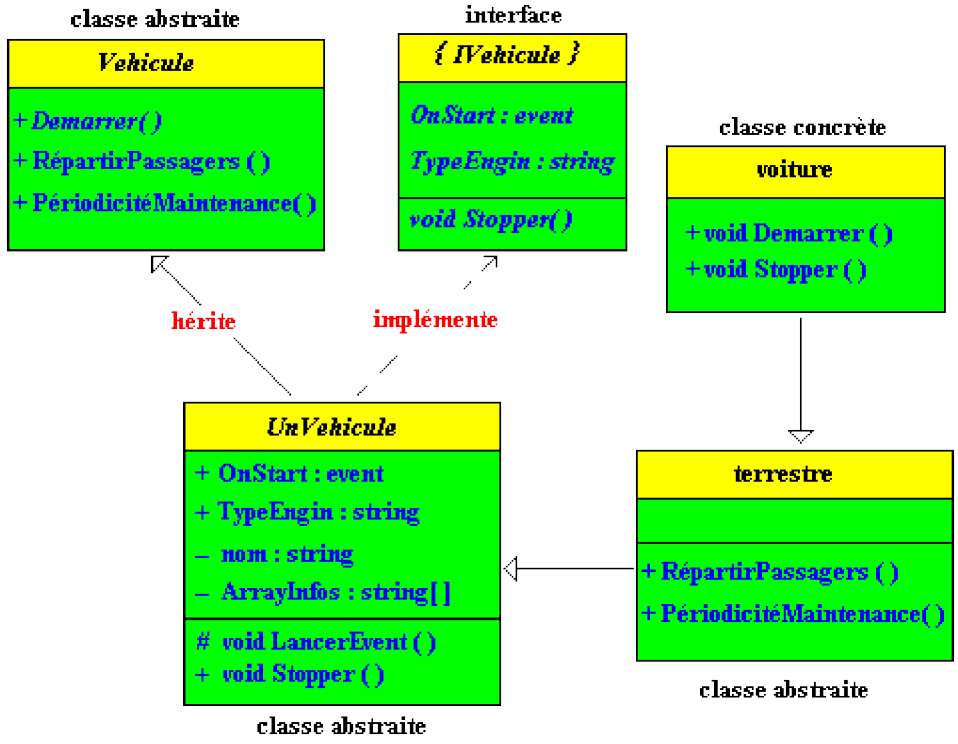
```
abstract class  Terrestre  :  UnVehicule {
  ....
  public override  string  TypeEngin     { // redéfinition de propriété
   get   {   return  base .TypeEngin ;  }
   set   {   string  nomTerre = value + "-Terrestre";
             base .TypeEngin = nomTerre ;  }
  }
}
```

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   { // instanciation 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 :

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
```

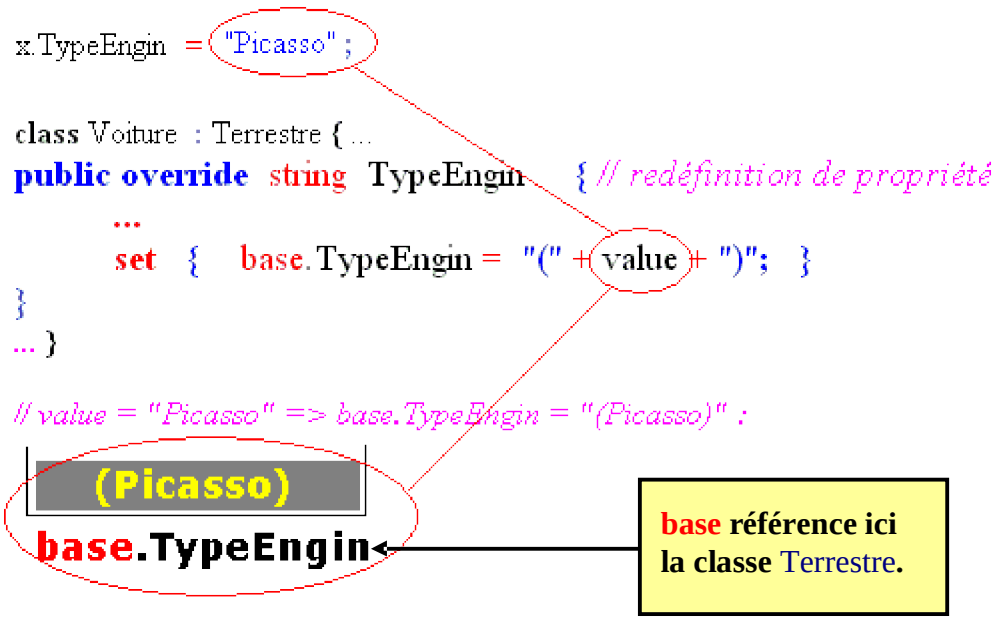
- **propriété TypeEngin dans la classe Terrestre :**

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

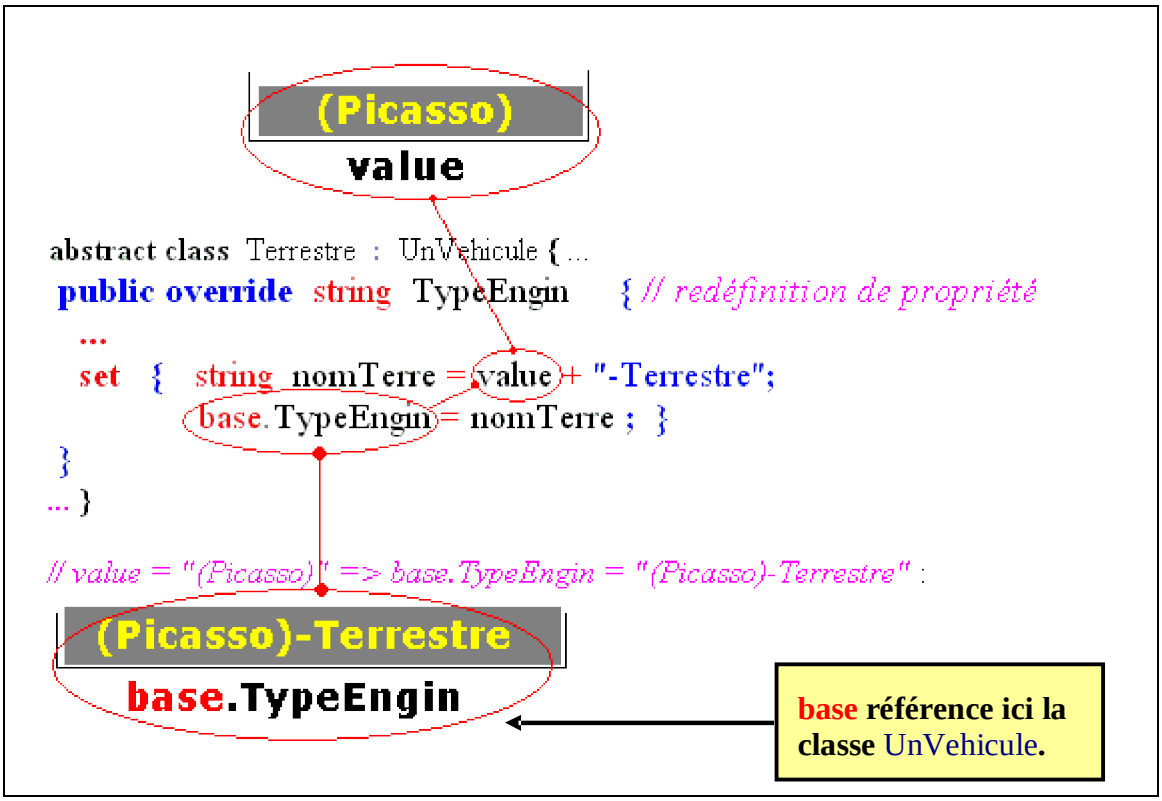
- **propriété TypeEngin dans la classe UnVehicule :**

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

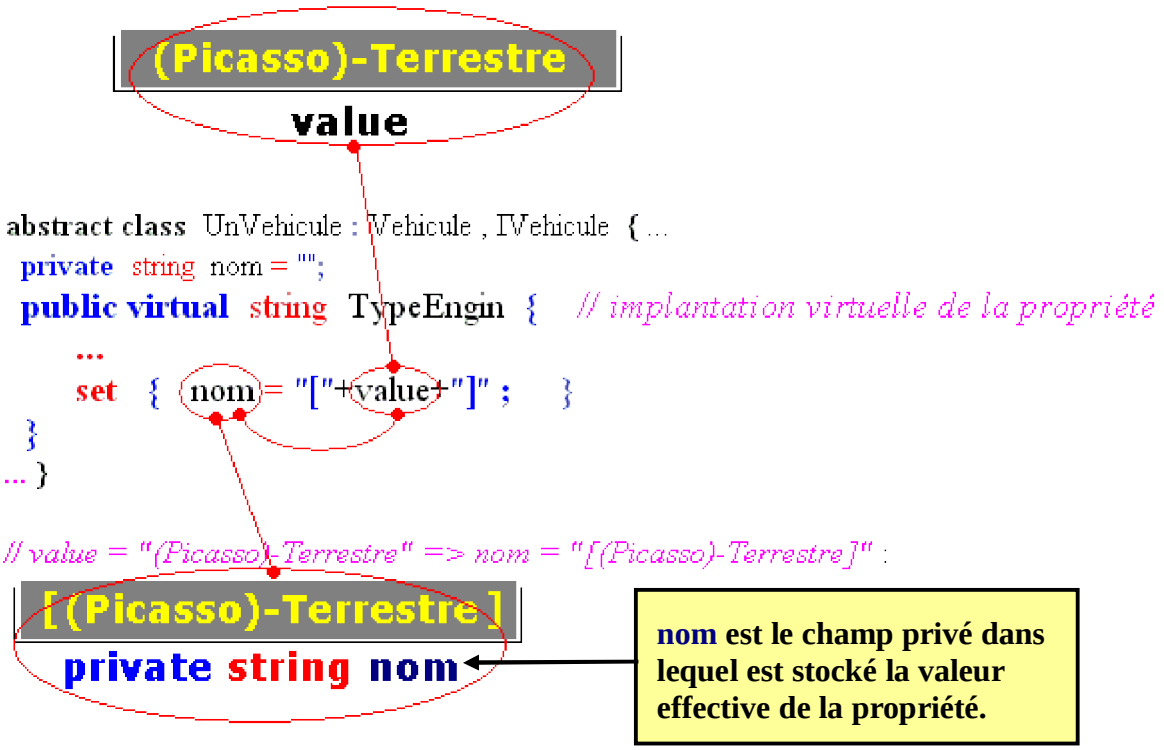
Définition de la propriété dans la classe Voiture (écriture) :



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


(Picasso)
value
abstract class Terrestre : UnVehicule { ...
public override string TypeEngin { // redéfinition de propriété
...
set { string nomTerre = value + "-Terrestre";
base.TypeEngin = nomTerre ; }
}
... }
// value = "(Picasso)" => base.TypeEngin = "(Picasso)-Terrestre" ;
(Picasso)-Terrestre
base.TypeEngin
base référence ici la classe UnVehicule.

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

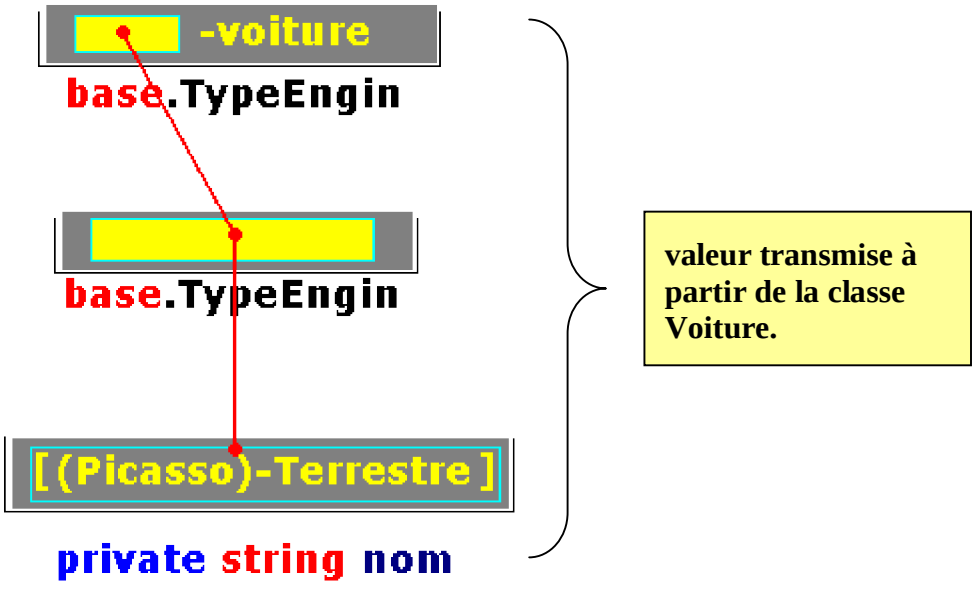


1.9.2 Détails de fonctionnement de la propriété TypeEngin en lecture

La propriété TypeEngin est en lecure dans :

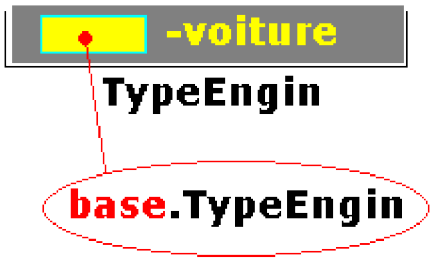
```
System.Console.WriteLine ( "x est une " + x.TypeEngin ) ;
```

Pour aller chercher la valeur effective, 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.



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

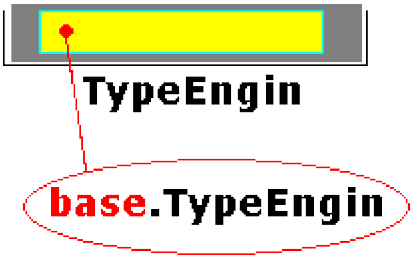
```
class Voiture  : Terrestre { ...
  public override  string  TypeEngin     { // redéfinition de propriété
        get  {  return  base.TypeEngin + "-voiture"; }
       ...
  }
... }
```



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) :

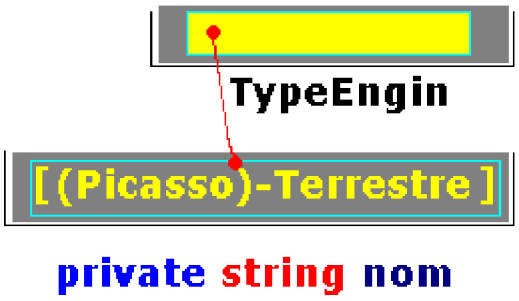
```
abstract class  Terrestre  :  UnVehicule { ...
  public override  string  TypeEngin     { // redéfinition de propriété
     get  {  return  base.TypeEngin ; }
     ...
  }
... }
```



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 ; }
      ...
  }
... }
```



L'accesseur get va chercher le résultat dans le champ nom.
nom est le champ privé dans lequel est stocké la valeur effective de la propriété.

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.

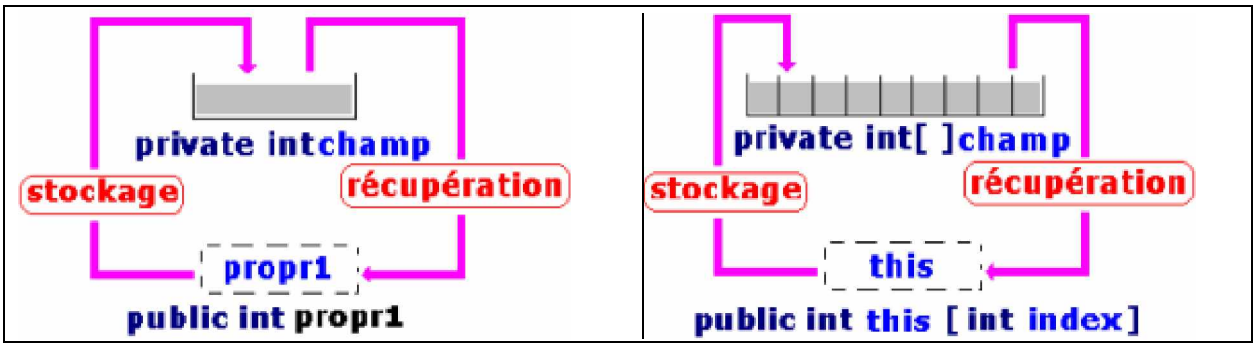
Tous les indexeurs sont représentés par l' opérateur **[ ]** . Les liens sur les propriétés ou les indexeurs du tableau ci-dessous renvoient directement au paragraphe associé.

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 **[ ]** . L'opérateur **[ ]** 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 Prop héritée est accessible par la syntaxe base.**Prop**	Un indexeur Prop 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

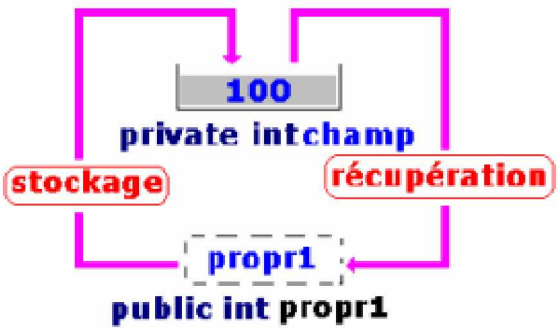
Propriété	Indexeur
Déclarée par son nom, avec champ de stockage : **private int** champ; **public int** propr1**{** **get { return** champ ; **}** **set {** champ = ***value*** ; **}** **}**	Déclaré par le mot clef **this**, avec champ de stockage : **private int** [ ] champ = **new int** [10]; **public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

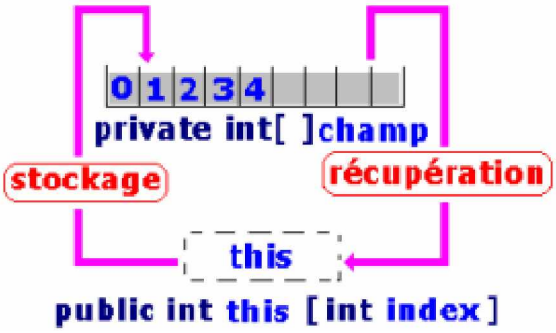


2.1.2 Utilisation

Propriété	Indexeur

Déclaration :
class clA **{**
  private int champ;
 	
  public int propr1**{**
  get { return champ ; **}**
  set { champ = ***value*** ; **}**
  }
}
Utilisation :
clA Obj = **new** clA( );
Obj.propr1 = 100 ;
 	
int x = Obj.prop1 ; *// x = 100*
	Déclaration :
	class clA **{**
	  private int [ ] champ = **new int** [10];
	
	  public int this [**int** index]**{**
	  get { return champ[index] ; **}**
	  set { champ[index] = ***value*** ; **}**
	  }
	}
	Utilisation :
	clA Obj = **new** clA( );
	for (**int** i =0; i<5; i++)
	  Obj[i] = i ;
	
	int x = Obj[2] ; *// x = 2*
	int y = Obj[3] ; *// x = 3*...





2.1.3 Paramètres

Propriété	Indexeur

Déclaration :

```
class clA {
  private int champ;

  public int propr1{
        get { return champ*10 ; }
        set { champ = value + 1  ; }
  }
}
```

value est un paramètre implicite.

Utilisation :

```
clA Obj = new clA( );
Obj.propr1 = 100 ;
int x = Obj.prop1 ;  // x = 1010
```

Déclaration :

```
class clA {
  private int [ ] champ = new int [10];

  public int this [int k]{
        get { return champ[ k ]*10 ; }
        set { champ[ k ] = value + 1  ; }
  }
}
```

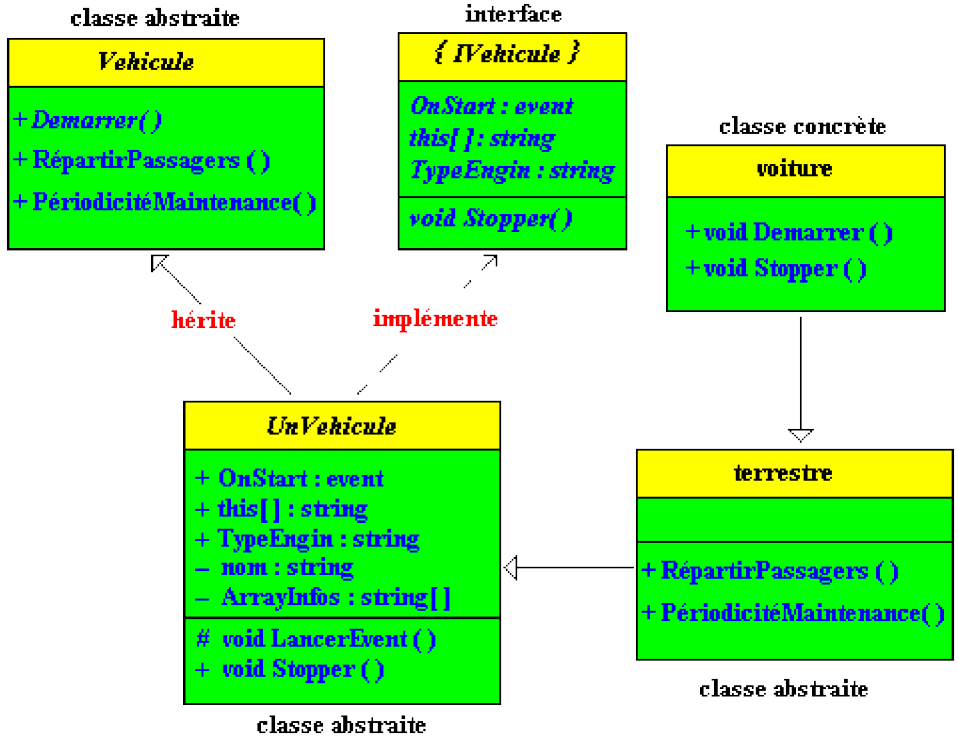
k est un paramètre formel de l'indexeur.

Utilisation :

```
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

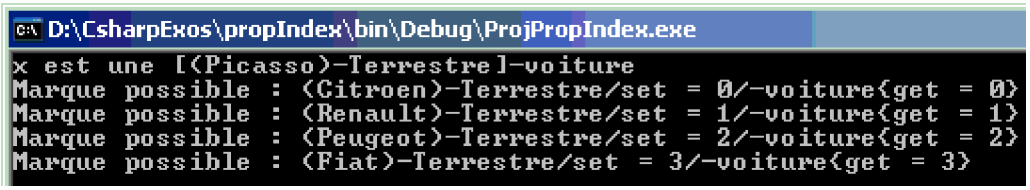
```
public delegate void  Starting ( );    // delegate declaration de type

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  )
 {
   // instanciation 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 :



Application fenêtrées et ressources en

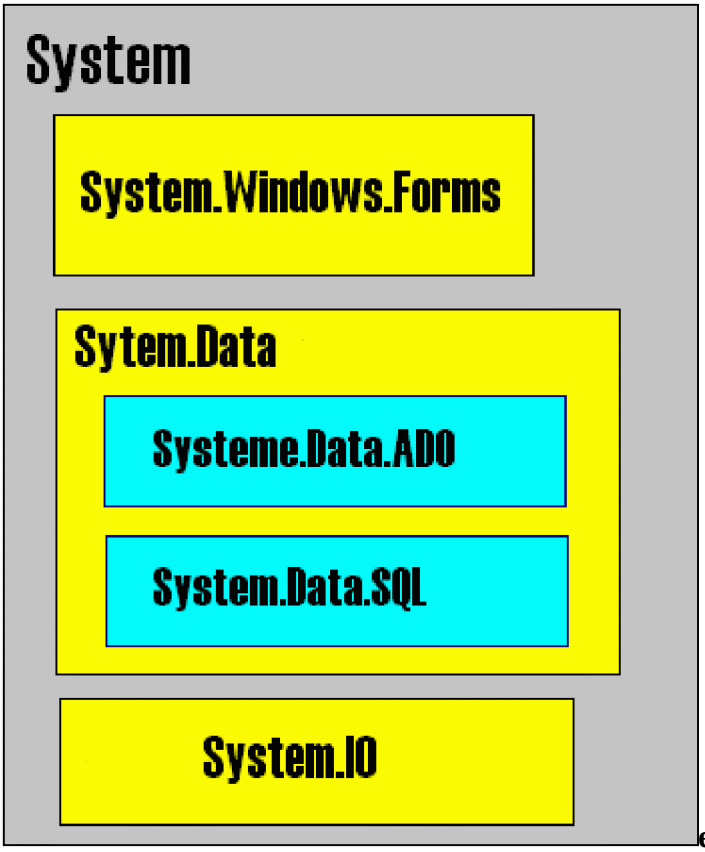
Plan général:

1. Les applications avec Interface Homme-Machine

1. Les applications avec Interface Homme-Machine

Les exemples et les traitement qui suivent sont effectués sous l'OS windows à partir de la version NetFramWork 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 :



etc ...

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.

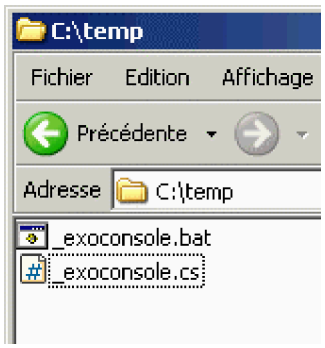
La classe **Form** de l'espace des noms **System.Windows.Forms** permet de créer une fenêtre classique avec barre de titre, zone client, boutons de fermeture, de zoom...

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

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

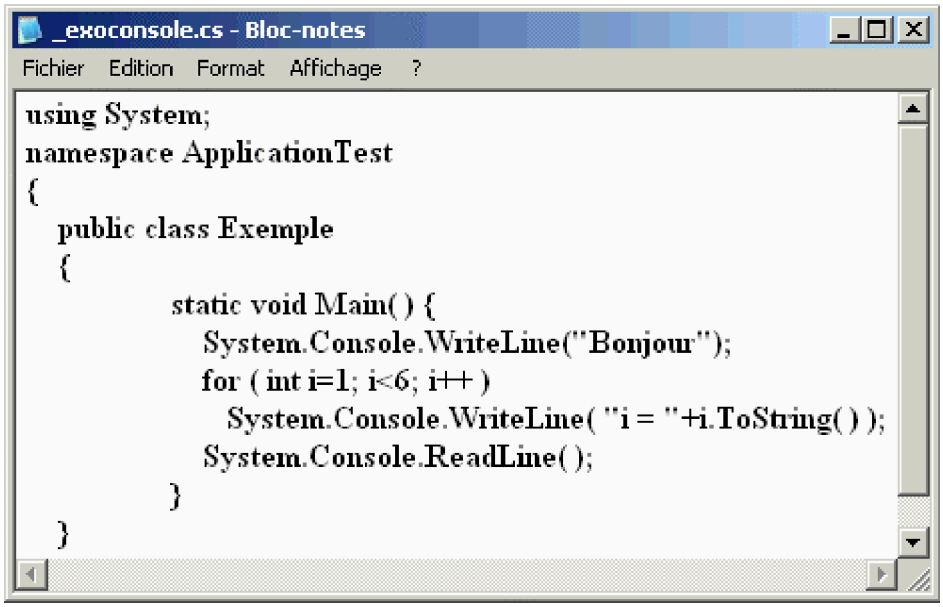
Construction de la commande de compilation "**_exoconsole.bat**" :

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'accéser 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)

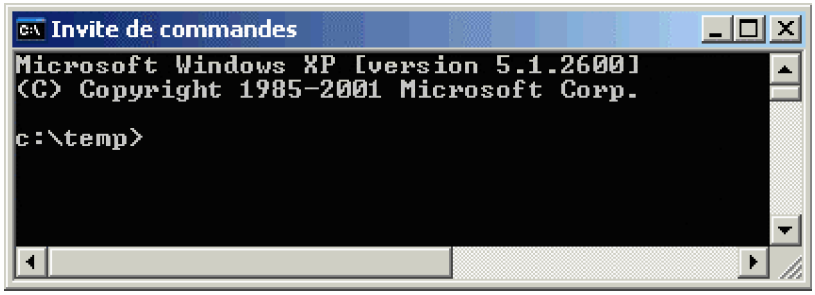
Texte de la commande dans le Bloc-note :

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



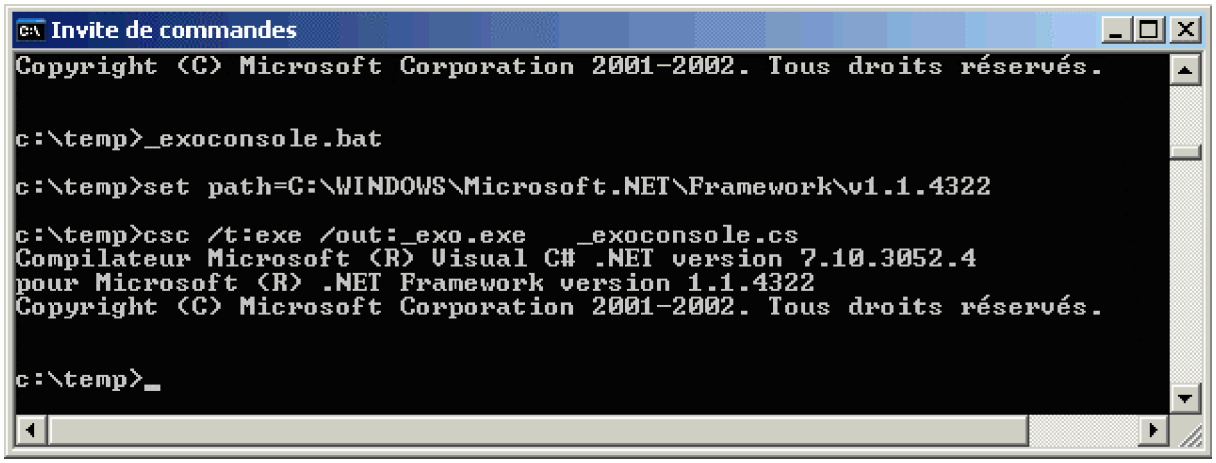
Le programme précédent affiche le mot Bonjour suivit de l'exécution de la boucle sur 5 itérations.

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

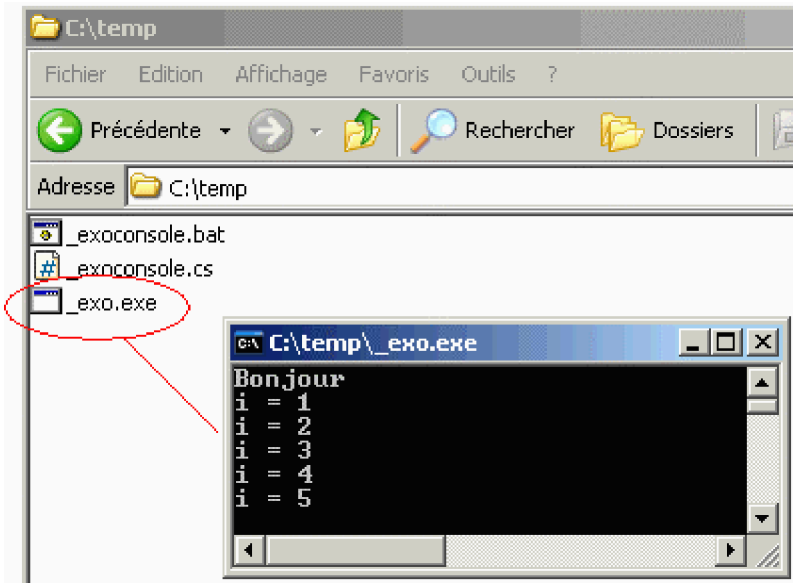


On tape au clavier et l'on exécute la commande "**_exoconsole.bat**" :

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



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

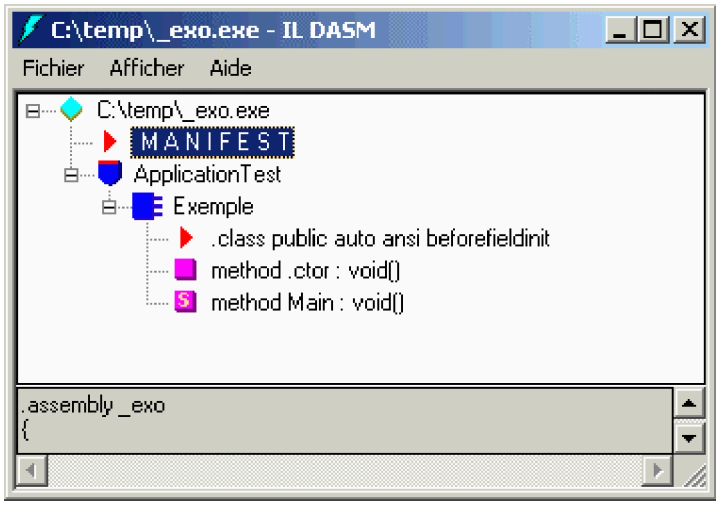


Nous constatons que l'exécution par le CLR du fichier **_exo.exe** a produit le résultat escompté c'est à dire l'affichage du mot Bonjour suivit de l'exécution de la boucle sur 5 itérations.

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:  ldloca.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
```

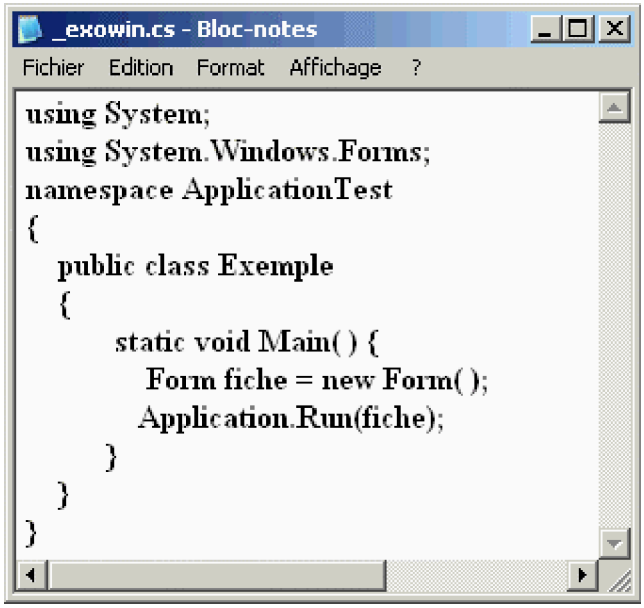
```
  IL_0029:  ldc.i4.6
  IL_002a:  blt.s      IL_000e


//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).

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( ) ;

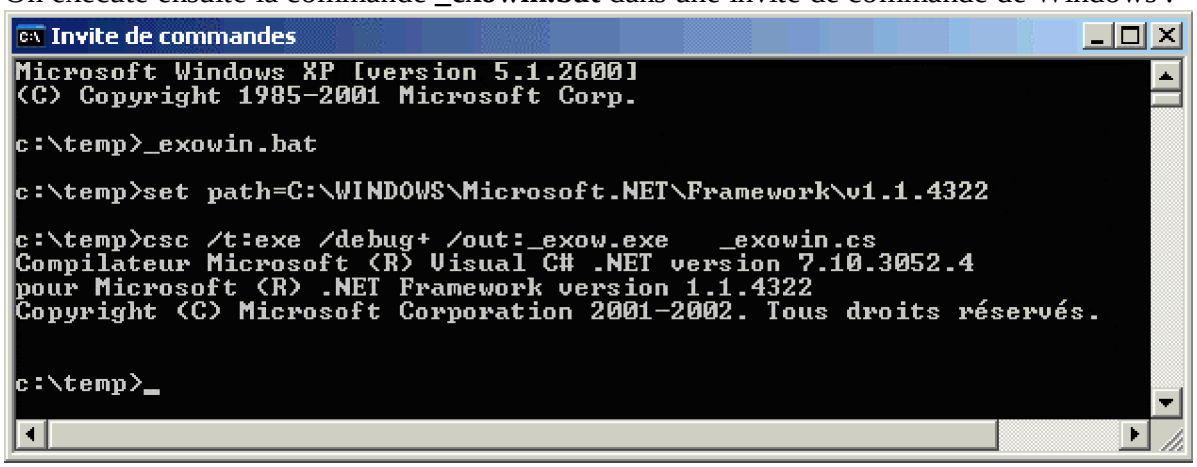
La fenêtre **fiche** est ensuite "initialisée" et "lancée" par l'instruction
Application.Run(fiche) ;

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.

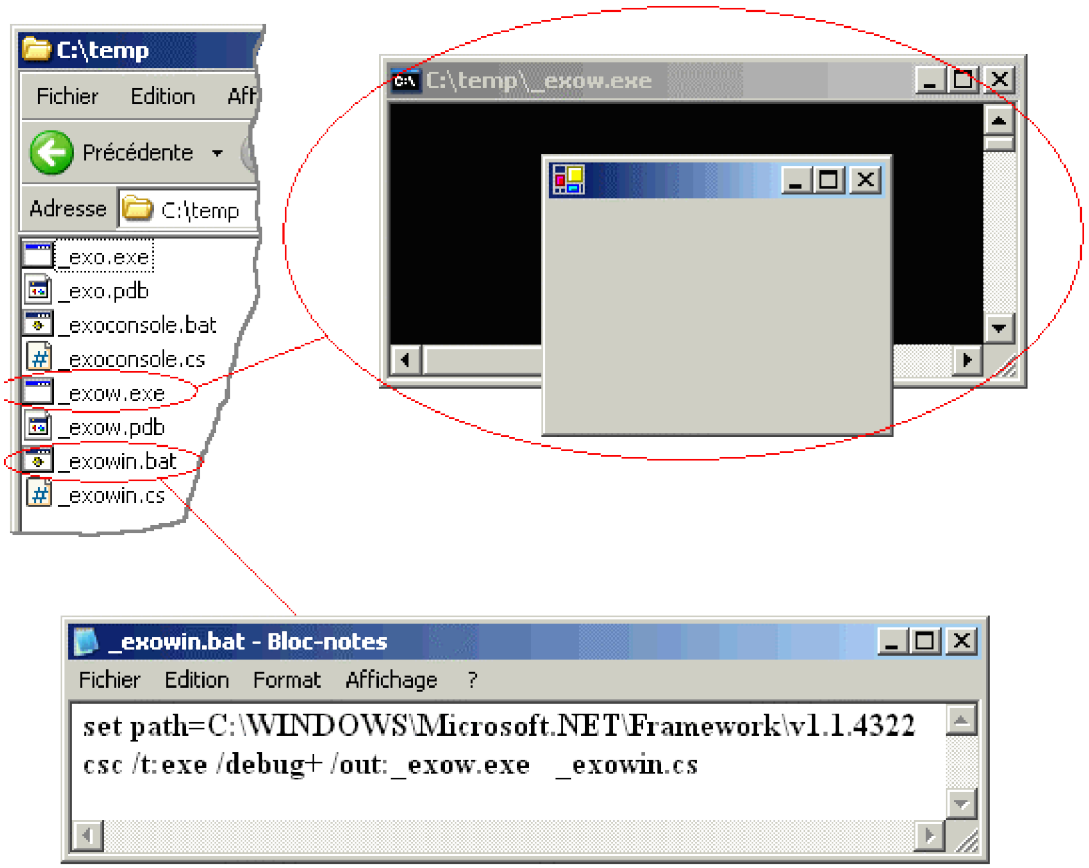
contenu fichier de commande **_exowin.bat** :

```
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 :



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.

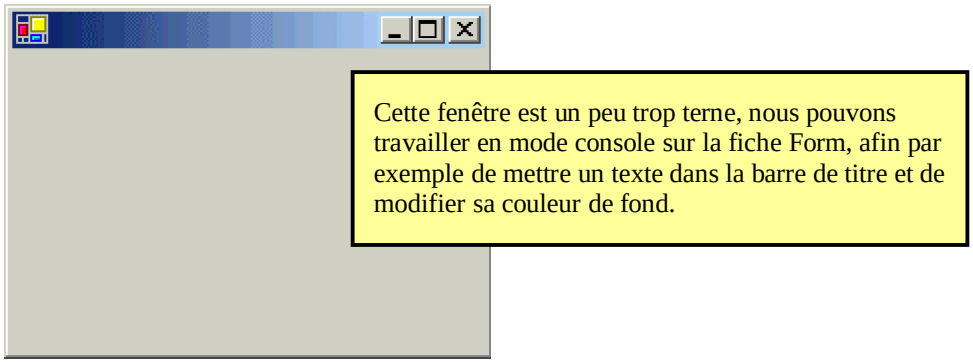
1.2.2 Fenêtre personnalisée sans fenêtre console

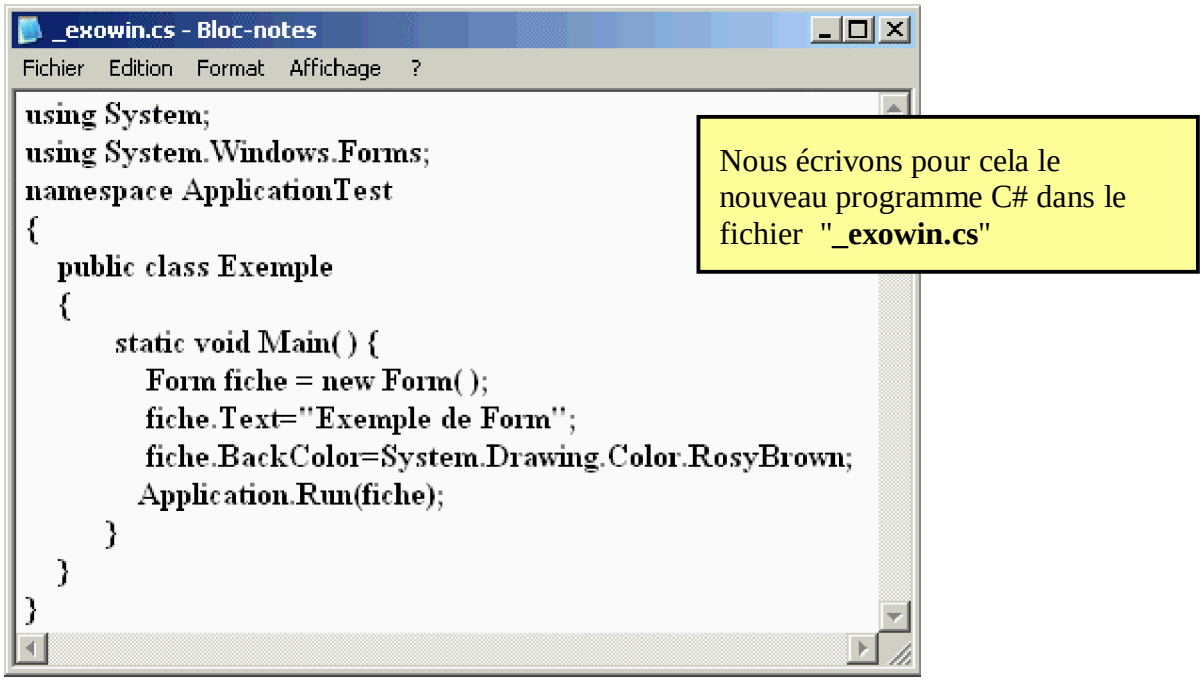
Si nous ne voulons pas voir apparaître de fenêtre de console mais seulement la fenêtre fiche, il faut alors changer dans le paramétrage du compilateur l'attribut **target.** De la valeur **csc /t:exe** il faut passer à la valeur **csc /t:winexe** :

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) :





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 :

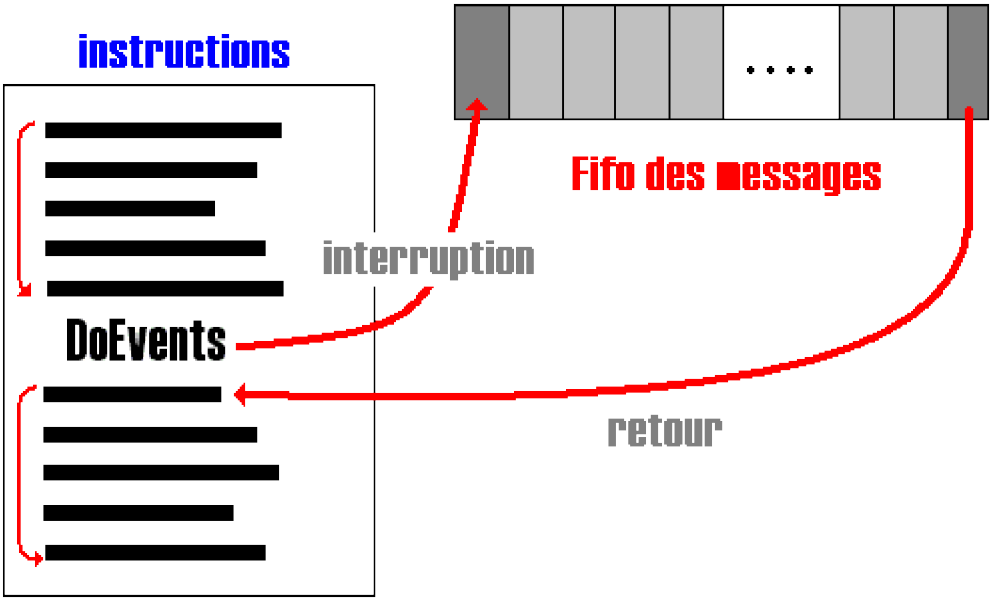
Ligne d'instruction du programme	Que fait le CLR
{	il initialise l'exécution de la méthode main
Form fiche = **new** Form( );	il instancie une Form nommée fiche
fiche.Text="Exemple de Form";	il met un texte dans le titre de fiche
fiche.BackColor=System.Drawing.Color.RosyBrown;	il modifie la couleur du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

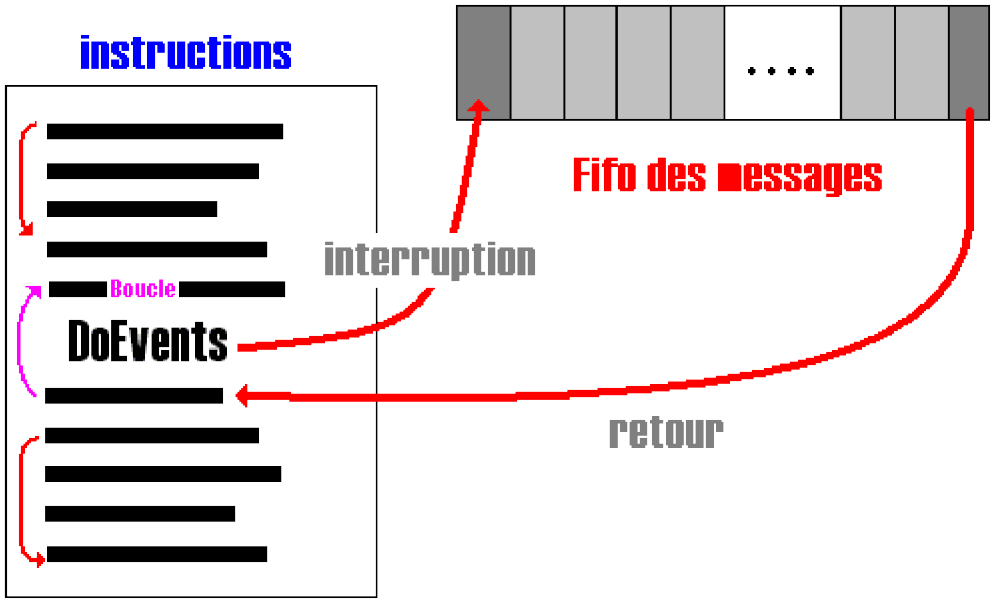
1.2.4 Que faire avec Application.DoEvents( )

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 créeons 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.

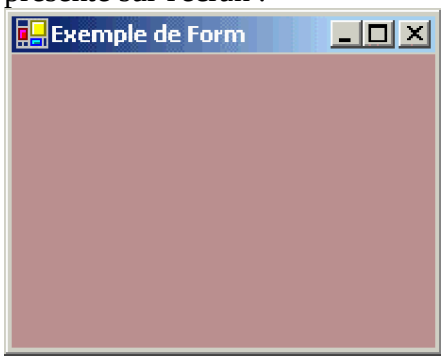
Choisissons une telle action par exemple lorsque l'utilisateur clique sur le bouton de fermeture

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

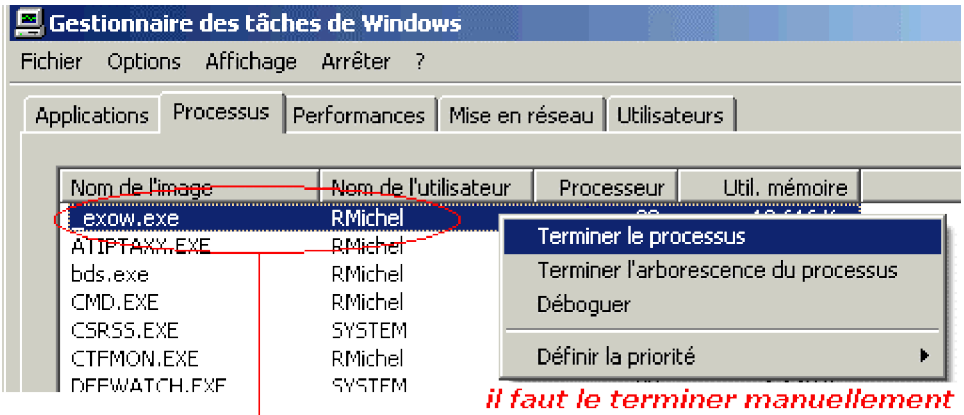
utilisable dans ce cas, en effet cette méthode arrête toutes les boucles de messages en cours sur tous les threads et ferme toutes les fenêtres de l'application; mais cette méthode ne force pas la fermeture de l'application.

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. *(prix très réduit pour l'éducation)*
- **C# Builder de Borland** reprenant les fonctionnalités de Visual C# dans Visual Studio, avec un intérêt supplémentaire pour un étudiant ou un apprenant : une version personnelle **gratuite** est téléchargeable (inclus dans Borland studio depuis 2005).
- **Le monde de l'open source** construit un produit nommé **sharpDevelop** qui fournit à tous gratuitement, les mêmes fonctions, mais en retard sur les versions de Microsoft.

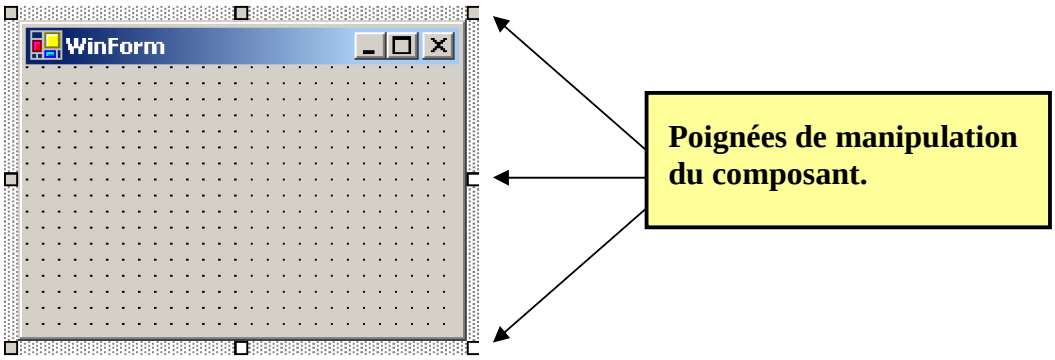
1.3 Un formulaire en C# est une fiche

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

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 studio 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 du RAD C# Builder (inclus dans Delphi 2005 depuis Borland studio 2005) en version personnelle (très suffisante pour déjà écrire de bonnes applications) nous illustrerons tous nos exemples avec ce RAD.

La fiche elle-même est figurée par l'image ci-dessous retaillable à volonté à partir de cliqué glissé sur l'une des huits 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 :

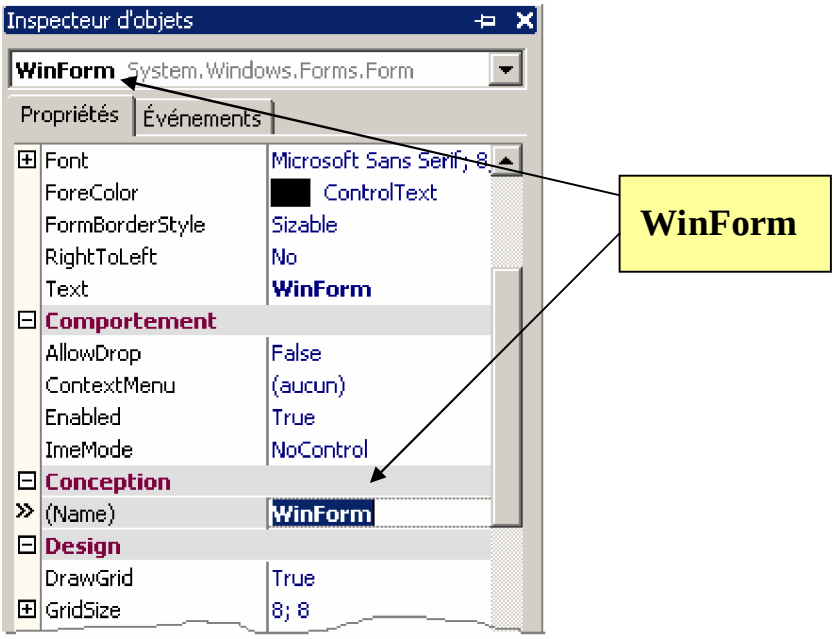
```
public Enum FormBorderStyle {Fixed3D, FixedDialog, FixedSingle, FixedToolWindow,
None, Sizable, SizableToolWindow  }
```

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

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

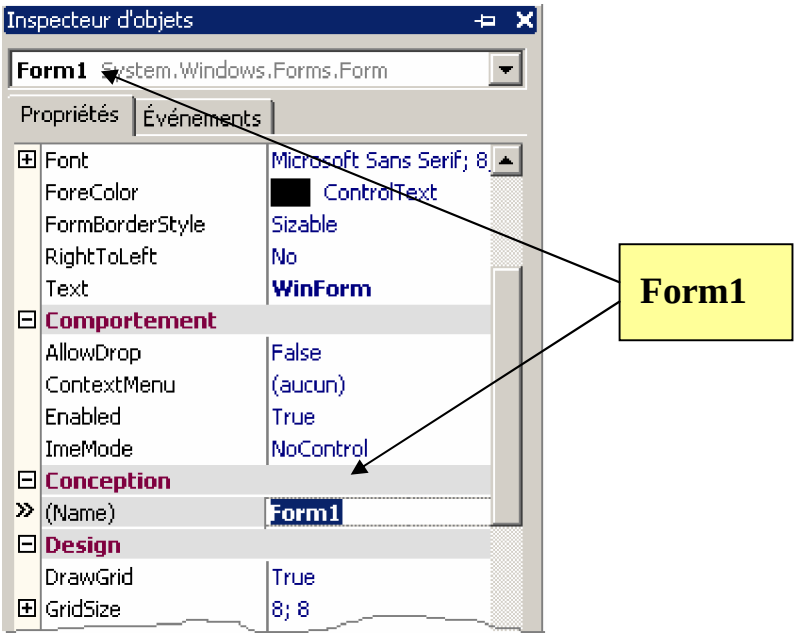
Toutes les propriétés en lecture et écriture d'une fiche sont accessibles à travers l'inspecteur d'objet qui répercute immédiatement en mode conception toute modification. Certaines provoquent des changements visuels d'autres non :

1°) changeons le nom d'identificateur de notre fiche dans le programme en modifiant la propriété Name qui vaut par défaut **WinForm** :

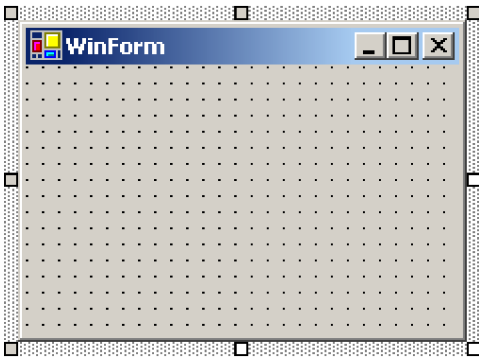


Le RAD construit automatiquement notre fiche principale comme une classe héritée de la classe Form et l'appelle WinForm :
public class WinForm : System.Windows.Forms.Form
{ ... }

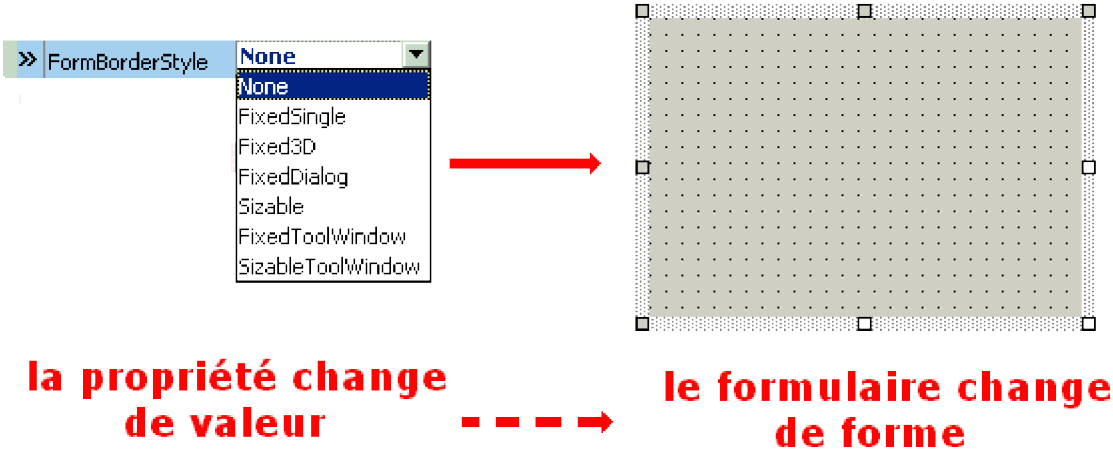
Après modification de la propriété Name par le texte **Form1**, nous obtenons :



La classe de notre formulaire s'appelle désormais **Form1**, mais son aspect visuel est resté le même :



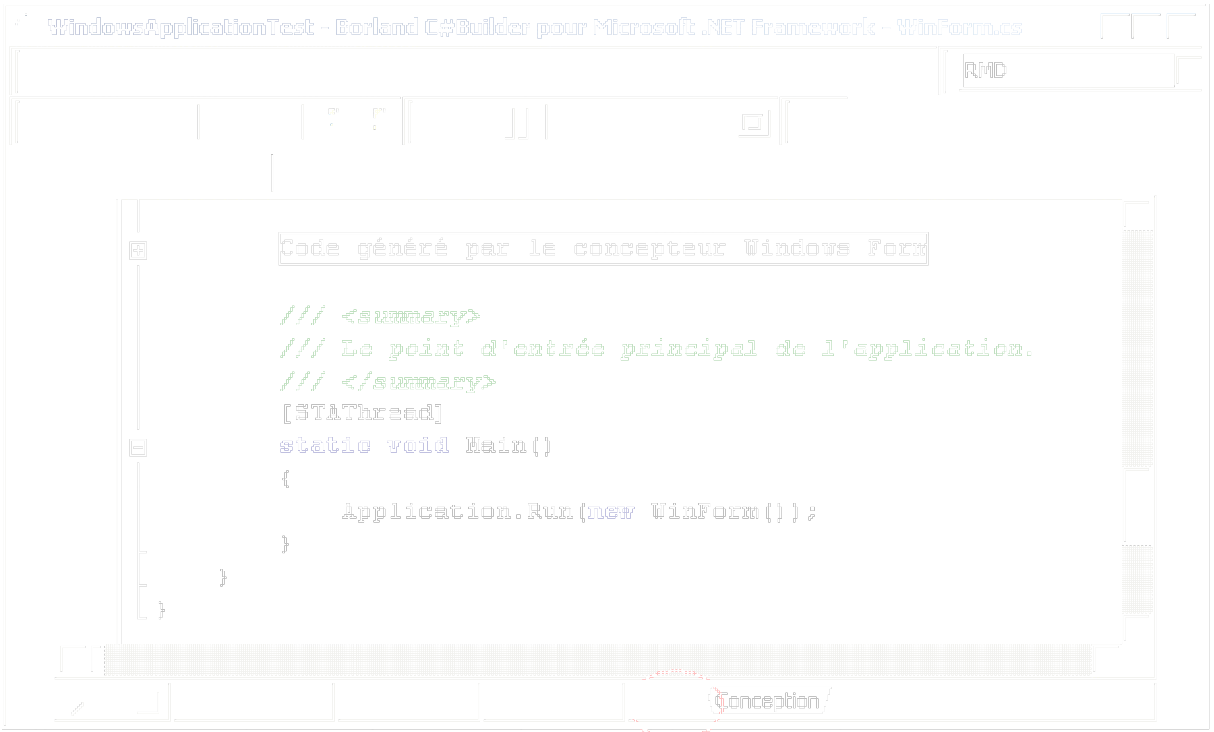
2°) Par exemple sélectionnons dans l'inspecteur d'objet de C# Builder, la propriété FormBorderStyle *(le style par défaut est* FormBorderStyle.Sizable *)* modifions la à la valeur **None** et regardons dans l'onglet conception la nouvelle forme de la fiche :



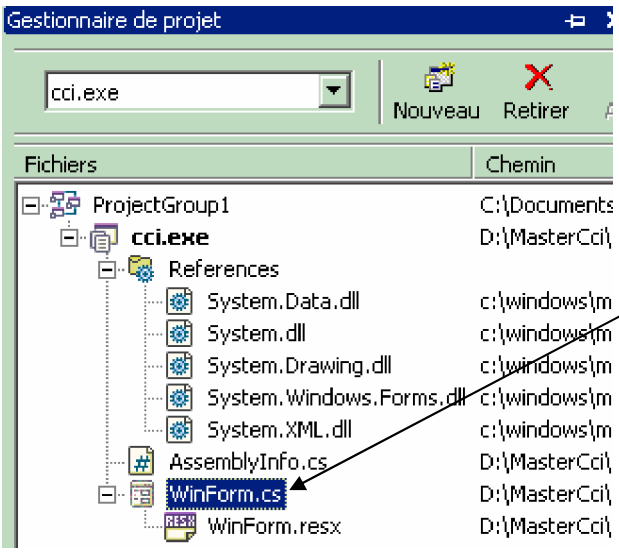
L'aspect visuel du formulaire a changé.

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 sa valeur initiale WinForm, voyons maintenant en supposant avoir appelé notre application **ProjApplication0** ce que C# Builder a engendré comme code source que nous trouvons dans l'onglet code pour notre formulaire :



Le RAD permet de visualiser le code source C# du formulaire



L'intégralité du code proposé par C# Builder est sauvegardé dans un fichier nommé WinForm.cs

Fichier WinForm.cs

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

namespace ProjApplication0
{

   public class WinForm : System.Windows.Forms.Form
   {

   }

}
```

Allure général du contenu de ce fichier affiché dans l'onglet code.

Fichier WinForm.cs

```
namespace ProjApplication0
{
    public class WinForm  : System.Windows.Forms.Form
    {
       public WinForm ()
       {
          ...
       }

       protected override void Dispose (bool disposing)
       { ...
       }

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

    }

}
```

La classe WinForm contient en première analyse 3 méthodes.

Premier éléments explicatifs de l'analyse du code :

La méthode	correspond
public WinForm ( ) **{** ... **}**	au constructeur d'objet de la classe WinForm et que la méthode
static void Main( ) **{** Application.Run (**new** WinForm ( )); **}**	au 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** WinForm ( )"de classe WinForm passé en paramètre à la méthode Run : c'est la fiche principale de l'application.
Application.Run (**new** WinForm ( )); 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 **WinForm** situé dans l'onglet code on se rend compte qu'il existe une ligne en grisé entre la méthode **Dispose** et la méthode **main** :

```
region Code généré par le concepteur Windows Form
```

Il s'agit en fait de code replié (masqué).

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

```
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 class WinForm : System.Windows.Forms.Form
   {
      /// <summary>
      /// Variable requise par le concepteur.
      /// </summary>
      private System.ComponentModel.Container components = null;

      public WinForm ( )
      {
         //
         // Requis pour la gestion du concepteur Windows Form
         //
         InitializeComponent( );
         //
         // TODO: Ajouter tout le code du constructeur après l'appel de InitializeComponent
```

```
        //
    }
    /// <summary>
    /// Nettoyage des ressources utilisées.
    /// </summary>
    protected override void Dispose (bool disposing)
    {
       if (disposing)  {
          if (components != null) {
             components.Dispose( );
          }
       }
       base.Dispose(disposing);
    }

    region Code généré par le concepteur Windows Form

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

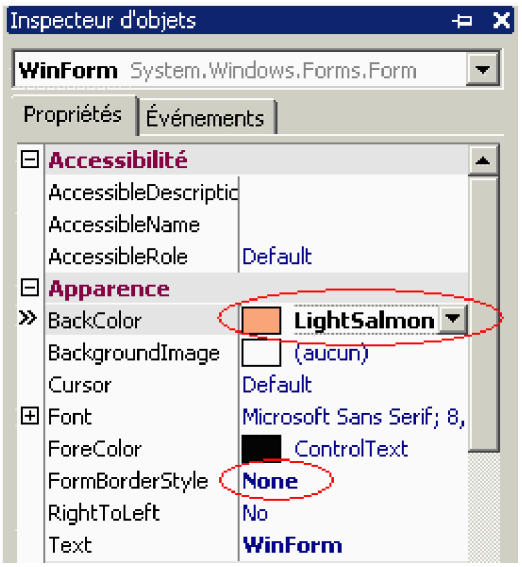
Si nous le déplions et nous voyons apparaître la méthode privée InitializeComponent( ) contenant du code qui a manifestement été généré directement. En outre cette méthode est appelée dans le constructeur d'objet WinForm :

```
public class WinForm  : System.Windows.Forms.Form
{
    public WinForm()
    {
       InitializeComponent();
    } ••• votre code

    protected override void Dispose (bool disposing)
    {  •••
    }

 # region Code généré par le concepteur Windows Form
    private void InitializeComponent()
    {
       code généré
       automatiquement
    }
 # region Code

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

Nous déplions la région Code généré par le concepteur Windows Form :

```
#region Code généré par le concepteur Windows Form
      /// <summary>
      /// Méthode requise pour la gestion du concepteur - ne pas modifier
      /// le contenu de cette méthode avec l'éditeur de code.
      /// </summary>
      private void InitializeComponent( )
      {
         //
         // WinForm
         //
         this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
         this.ClientSize = new System.Drawing.Size(232, 157);
         this.Name = "WinForm";
         this.Text = "WinForm";
      }
#endregion
```

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 Code généré par le concepteur Windows Form
   /// <summary>
   /// Méthode requise pour la gestion du concepteur - ne pas modifier
   /// le contenu de cette méthode avec l'éditeur de code.
   /// </summary>
   private void InitializeComponent( )
   {
      //
```

```
            // WinForm
            //
            this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
            this.BackColor = System.Drawing.Color.LightSalmon;
            this.ClientSize = new System.Drawing.Size(232, 157);
            this.FormBorderStyle = System.Windows.Forms.FormBorderStyle.None;
            this.Name = "WinForm";
            this.Text = "WinForm";
        }
        #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)  {
            if (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 gére 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 influer 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.**

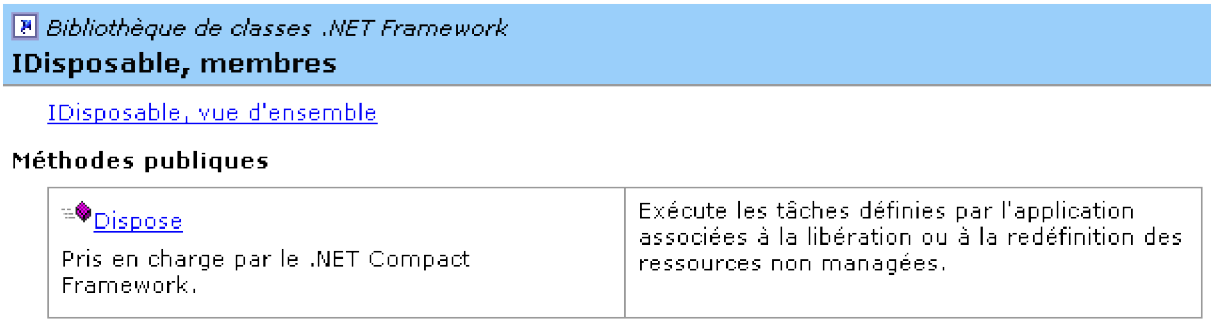
Microsoft propose des conseils pour écrire la méthode Dispose :

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



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 libérér 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 réellemnt 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)  { // Nettoyage des ressources managées
            if (components != null) {
               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 class WinForm : System.Windows.Forms.Form
   {
      /// <summary>
       /// Variable requise par le concepteur.
       /// </summary>
      private System.ComponentModel.Container components = null;
      private UnComposant MonComposant = new UnComposant( );


      public WinForm ( )
      {

         InitializeComponent( );

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


      region Code généré par le concepteur Windows Form

      /// <summary>
      /// Le point d'entrée principal de l'application.
      /// </summary>
      [STAThread]
```

Déclaration-instanciation d'un composant personnel.

Ajout du composant personnel dans la Fifo de components.

Notre composant personnel est libéré avec les autres

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

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 button1de 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 WinForm ( ));
}
```

Cet attribut placé ici devant la méthode **main** qualifie la manière dont le CLR exécutera l'application, il signifie : **S**ingle **T**hread **A**partments.

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] M**ulti **T**hread **A**partments, 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 WinForm ( ));
}
```

Des contrôles dans les formulaires

Plan général:

1. Les contrôles et les fonds graphiques

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 aquis une dextérité minimale dans cette manimulation visuelle (déposer des composants, utiliser l'inspecteur d'objet (ou inspecteur de propriétés), modidifer 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 !

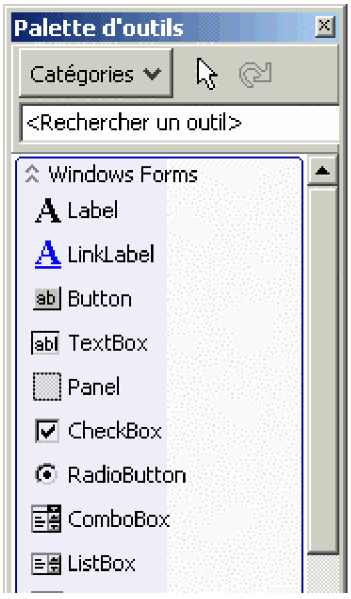
1.1 les composants en général

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 deux RAD Visual C# et C#Builder 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.

C# Builder :



Visual C# :

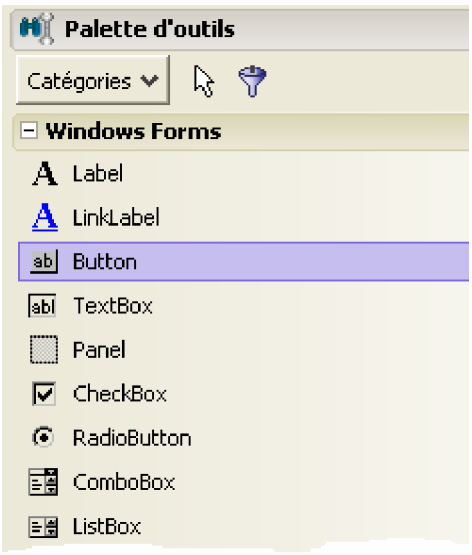
Cette distinction entre visuel et non visuel n'est pas très précise et dépend de la présentation dans le RAD. Cette variabilité de la dénomination n'a aucune importance pour l'utilisateur car tous les composants ont un fonctionnement du code identique dans les deux RAD.

En effet si nous prenons les classes « Label » et « Button » :

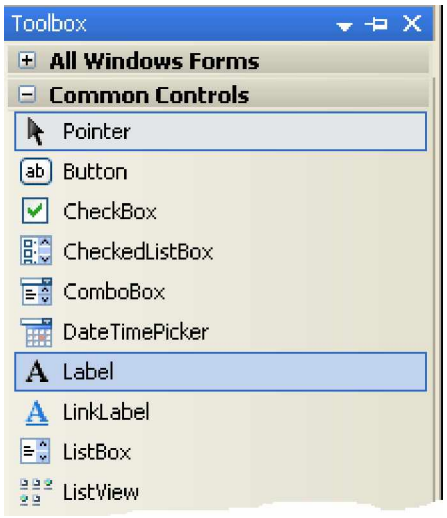
System.ComponentModel.Component
System.Windows.Forms.Label
System.Windows.Forms.Label

Nous voyons que Visual C# range « Label » et « Button » dans les contrôles alors que C# Builder les range dans les Windows Forms. Ci-dessous les outils de composants proposés par les deux RAD :

C# Builder 2005



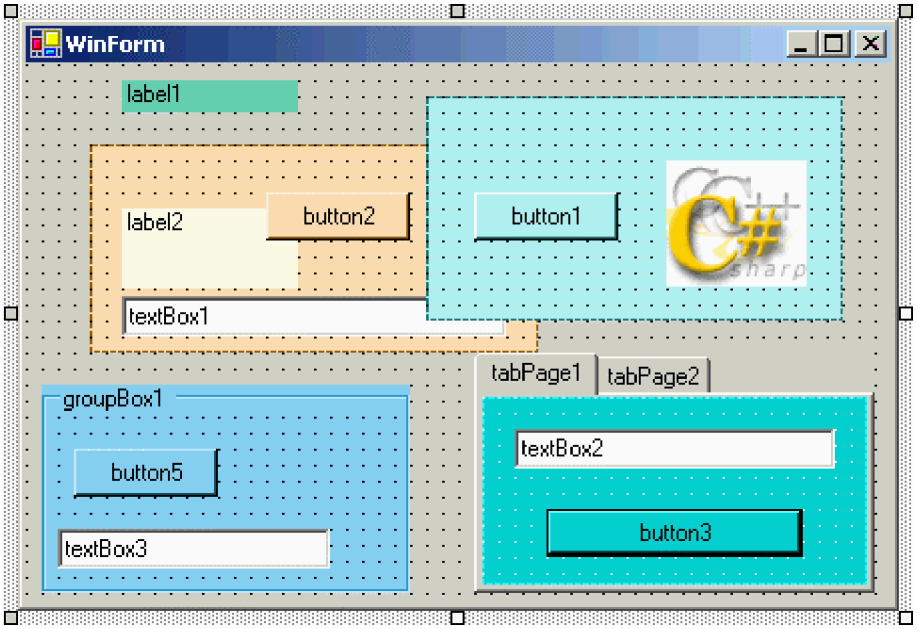
Visual C# 2005 :



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

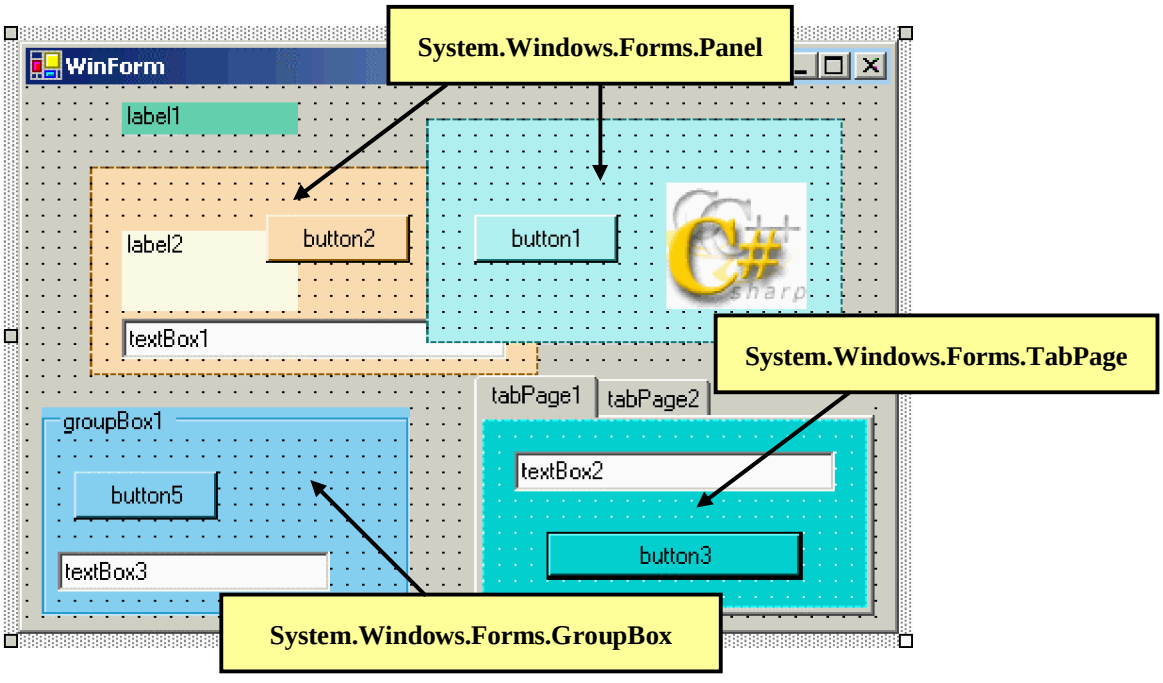
Voici un exemple de fiche comportant 7 catégories de contrôles différents :



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

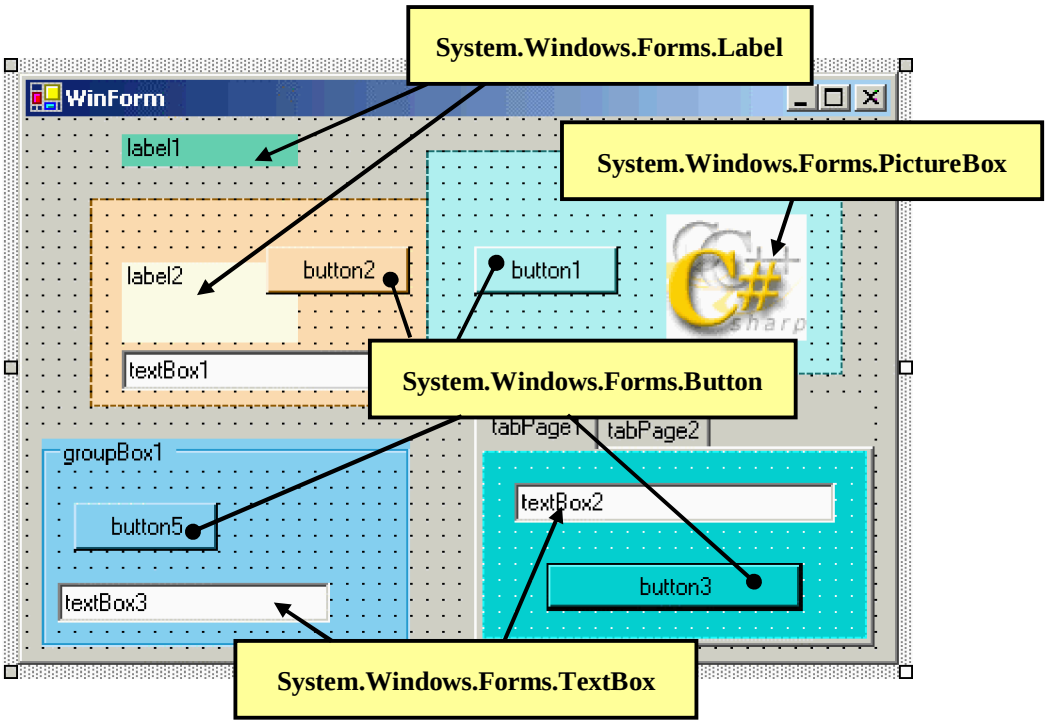
Sur la fiche précédente nous relevons outre le formulaire lui-même, quatre conteneurs visuels répartis en trois catégories de conteneurs visuels :



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 :



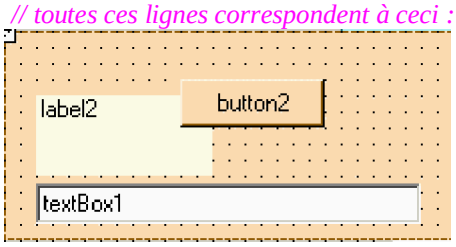
Le code C# engendré pour cette interface :

```
public class  WinForm  :  System .Windows.Forms.Form
{
 /// <summary>
 /// Variable requise par le concepteur.
 /// </summary>
 private  System .ComponentModel.Container components  =  null;
 private  System .Windows.Forms.Panel panel1 ;
 private  System .Windows.Forms.Label label1 ;
 private  System .Windows.Forms.Label label2 ;
 private  System .Windows.Forms.Panel panel2 ;
 private  System .Windows.Forms.PictureBox pictureBox1 ;
 private  System .Windows.Forms.Button button1 ;
 private  System .Windows.Forms.TextBox textBox1 ;
 private  System .Windows.Forms.Button button2 ;
 private  System .Windows.Forms.GroupBox groupBox1 ;
 private  System .Windows.Forms.TabControl tabControl1 ;
 private  System .Windows.Forms.TabPage tabPage1 ;
```

```
 private  System .Windows.Forms.TabPage tabPage2 ;
 private  System .Windows.Forms.Button button3 ;
 private  System .Windows.Forms.TextBox textBox2 ;
 private  System .Windows.Forms.ListBox listBox1 ;
 private  System .Windows.Forms.Button button4 ;
 private  System .Windows.Forms.Button button5 ;
 private  System .Windows.Forms.TextBox textBox3 ;
 private  System .Windows.Forms.MainMenu mainMenu1 ;

 public  WinForm ( ) {
   //
  // Requis pour la gestion du concepteur Windows Form
  //
   InitializeComponent ( );

   //
  // TODO: Ajouter tout le code du constructeur après l'appel de InitializeComponent
  //
  }

  /// <summary>
 /// Nettoyage des ressources utilisées.
 /// </summary>
  protected override void  Dispose  ( bool  disposing ) {
  if ( disposing )  {
   if ( components  !=  null)   {
     components.Dispose ( );
   }
  }
   base .Dispose ( disposing );
 }

   #region Code généré par le concepteur Windows Form
private void  InitializeComponent ( )
{
System .Resources.ResourceManager resources  =
        new  System .Resources.ResourceManager ( typeof ( WinForm ));
  this.panel1  =  new  System .Windows.Forms.Panel ( );
  this.button2  =  new  System .Windows.Forms.Button ( );
  this.textBox1  =  new  System .Windows.Forms.TextBox ( );
  this.label2  =  new  System .Windows.Forms.Label ( );
  this.label1  =  new  System .Windows.Forms.Label ( );
  this.panel2  =  new  System .Windows.Forms.Panel ( );
  this.button1  =  new  System .Windows.Forms.Button ( );
  this.pictureBox1  =  new  System .Windows.Forms.PictureBox ( );
  this.groupBox1  =  new  System .Windows.Forms.GroupBox ( );
  this.textBox3  =  new  System .Windows.Forms.TextBox ( );
  this.button5  =  new  System .Windows.Forms.Button ( );
  this.tabControl1  =  new  System .Windows.Forms.TabControl ( );
  this.tabPage1  =  new  System .Windows.Forms.TabPage ( );
  this.textBox2  =  new  System .Windows.Forms.TextBox ( );
  this.button3  =  new  System .Windows.Forms.Button ( );
  this.tabPage2  =  new  System .Windows.Forms.TabPage ( );
  this.button4  =  new  System .Windows.Forms.Button ( );
  this.listBox1  =  new  System .Windows.Forms.ListBox ( );
  this.mainMenu1  =  new  System .Windows.Forms.MainMenu ( );
  this.panel1.SuspendLayout ( );
  this.panel2.SuspendLayout ( );
  this.groupBox1.SuspendLayout ( );
  this.tabControl1.SuspendLayout ( );
  this.tabPage1.SuspendLayout ( );
```

```
this.tabPage2.SuspendLayout ( );
this.SuspendLayout ( );
```

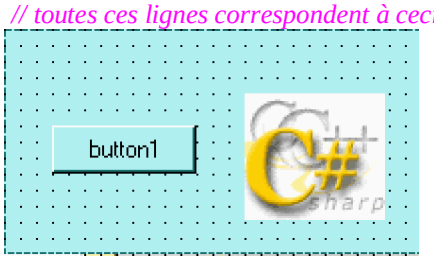
```
//
// panel1
//
this.panel1.BackColor  = System .Drawing.Color.NavajoWhite ;
this.panel1.Controls.Add (this .button2 );
this.panel1.Controls.Add (this .textBox1 );
this.panel1.Controls.Add (this .label2 );
this.panel1.Location  = new  System .Drawing.Point ( 32, 40 );
this.panel1.Name  = "panel1";
this.panel1.Size  = new  System .Drawing.Size ( 224, 104 );
this.panel1.TabIndex  = 0 ;
//
// button2
//
this.button2.Location  = new  System .Drawing.Point ( 88, 24 );
this.button2.Name  = "button2";
this.button2.Size  = new  System .Drawing.Size ( 72, 24 );
this.button2.TabIndex  = 4 ;
this.button2.Text  = "button2";
//
// textBox1
//
this.textBox1.Location  = new  System .Drawing.Point ( 16, 76 );
this.textBox1.Name  = "textBox1";
this.textBox1.Size  = new  System .Drawing.Size ( 192, 20 );
this.textBox1.TabIndex  = 3 ;
this.textBox1.Text  = "textBox1";
//
// label2
//
this.label2.BackColor  = System .Drawing.SystemColors.Info ;
this.label2.Location  = new  System .Drawing.Point ( 16, 32 );
this.label2.Name  = "label2";
this.label2.Size  = new  System .Drawing.Size ( 88, 40 );
this.label2.TabIndex  = 2 ;
this.label2.Text  = "label2";

// toutes ces lignes correspondent à ceci :
```




```
//
// panel2
//
this.panel2.BackColor  = System .Drawing.Color.PaleTurquoise ;
this.panel2.Controls.Add (this .button1 );
this.panel2.Controls.Add (this .pictureBox1 );
this.panel2.Location  = new  System .Drawing.Point ( 200, 16 );
this.panel2.Name  = "panel2";
this.panel2.Size  = new  System .Drawing.Size ( 208, 112 );
this.panel2.TabIndex  = 2 ;
//
```

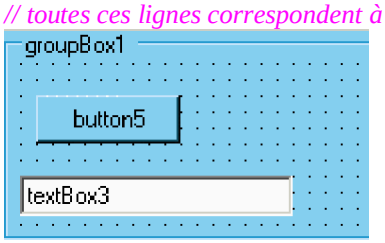
```
// button1
//
this.button1.Location = new System.Drawing.Point( 24, 48 );
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size( 72, 24 );
this.button1.TabIndex = 4;
this.button1.Text = "button1";
//
// pictureBox1
//
this.pictureBox1.BackColor = System.Drawing.Color.DarkKhaki;
this.pictureBox1.Image = ((System.Drawing.Image)(resources.GetObject
("pictureBox1.Image")));
// la ligne précédente charge l'image :
```

```
this.pictureBox1.Location = new System.Drawing.Point( 120, 32 );
this.pictureBox1.Name = "pictureBox1";
this.pictureBox1.Size = new System.Drawing.Size( 70, 63 );
this.pictureBox1.SizeMode = System.Windows.Forms.PictureBoxSizeMode.AutoSize;
this.pictureBox1.TabIndex = 0;
this.pictureBox1.TabStop = false;

// toutes ces lignes correspondent à ceci :
```

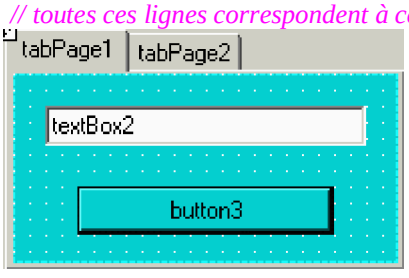



```
//
// groupBox1
//
this.groupBox1.BackColor = System.Drawing.Color.SkyBlue;
this.groupBox1.Controls.Add(this.textBox3);
this.groupBox1.Controls.Add(this.button5);
this.groupBox1.Location = new System.Drawing.Point( 8, 160 );
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size( 184, 104 );
this.groupBox1.TabIndex = 4;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "groupBox1";
//
// textBox3
//
this.textBox3.Location = new System.Drawing.Point( 8, 72 );
this.textBox3.Name = "textBox3";
this.textBox3.Size = new System.Drawing.Size( 136, 20 );
this.textBox3.TabIndex = 1;
this.textBox3.Text = "textBox3";
//
// button5
//
this.button5.Location = new System.Drawing.Point( 16, 32 );
this.button5.Name = "button5";
```

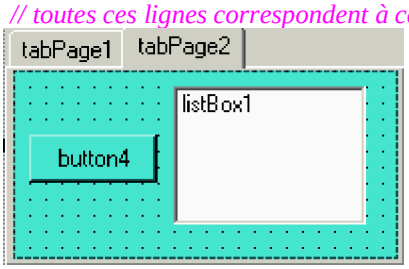
```
this.button5.Size = new System.Drawing.Size( 72, 24 );
this.button5.TabIndex = 0;
this.button5.Text = "button5";

// toutes ces lignes correspondent à ceci :
```




```
//
// tabControl1
//
this.tabControl1.Controls.Add(this.tabPage1);
this.tabControl1.Controls.Add(this.tabPage2);
this.tabControl1.Location = new System.Drawing.Point( 224, 144 );
this.tabControl1.Name = "tabControl1";
this.tabControl1.SelectedIndex = 0;
this.tabControl1.Size = new System.Drawing.Size( 200, 120 );
this.tabControl1.TabIndex = 5;
//
// tabPage1
//
this.tabPage1.BackColor = System.Drawing.Color.DarkTurquoise;
this.tabPage1.Controls.Add(this.textBox2);
this.tabPage1.Controls.Add(this.button3);
this.tabPage1.Location = new System.Drawing.Point( 4, 22 );
this.tabPage1.Name = "tabPage1";
this.tabPage1.Size = new System.Drawing.Size( 192, 94 );
this.tabPage1.TabIndex = 0;
this.tabPage1.Text = "tabPage1";
//
// textBox2
//
this.textBox2.Location = new System.Drawing.Point( 16, 16 );
this.textBox2.Name = "textBox2";
this.textBox2.Size = new System.Drawing.Size( 160, 20 );
this.textBox2.TabIndex = 1;
this.textBox2.Text = "textBox2";
//
// button3
//
this.button3.Location = new System.Drawing.Point( 32, 56 );
this.button3.Name = "button3";
this.button3.Size = new System.Drawing.Size( 128, 24 );
this.button3.TabIndex = 0;
this.button3.Text = "button3";

// toutes ces lignes correspondent à ceci :
```



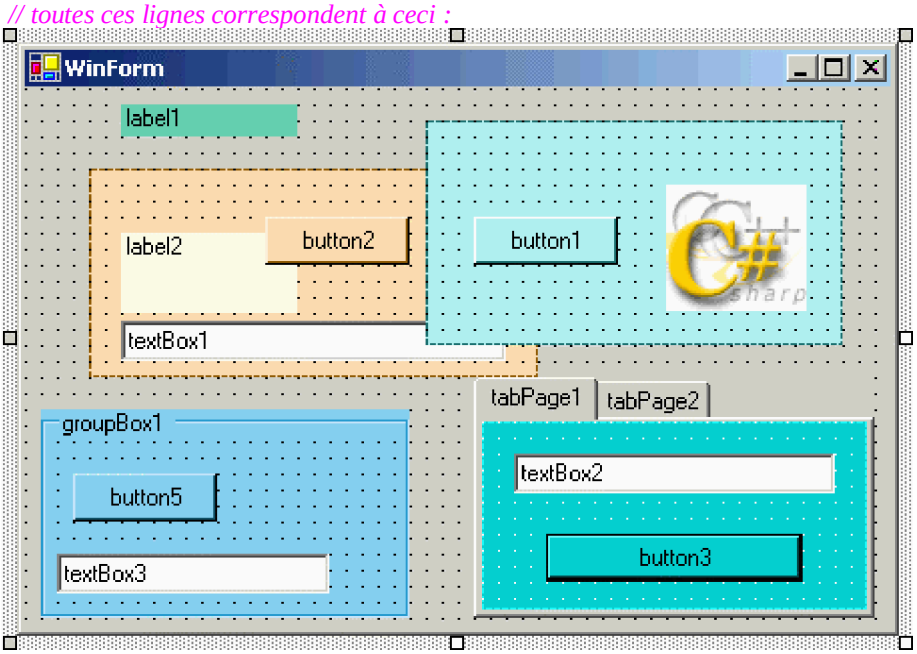
```
//
// tabPage2
//
this.tabPage2.BackColor = System.Drawing.Color.Turquoise;
this.tabPage2.Controls.Add(this.button4);
this.tabPage2.Controls.Add(this.listBox1);
this.tabPage2.Location = new System.Drawing.Point(4, 22);
this.tabPage2.Name = "tabPage2";
this.tabPage2.Size = new System.Drawing.Size(192, 94);
this.tabPage2.TabIndex = 1;
this.tabPage2.Text = "tabPage2";
//
// button4
//
this.button4.Location = new System.Drawing.Point(8, 32);
this.button4.Name = "button4";
this.button4.Size = new System.Drawing.Size(64, 24);
this.button4.TabIndex = 1;
this.button4.Text = "button4";
//
// listBox1
//
this.listBox1.Location = new System.Drawing.Point(80, 8);
this.listBox1.Name = "listBox1";
this.listBox1.Size = new System.Drawing.Size(96, 69);
this.listBox1.TabIndex = 0;

// toutes ces lignes correspondent à ceci :
```




```
//
// label1
//
this.label1.BackColor = System.Drawing.Color.MediumAquamarine;
this.label1.Location = new System.Drawing.Point(48, 8);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(88, 16);
this.label1.TabIndex = 1;
this.label1.Text = "label1";

// les 6 lignes précédentes correspondent à ceci :
```


```
//
// WinForm
//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(432, 269);
```

```
this.Controls.Add (this .tabControl1 );
this.Controls.Add (this .groupBox1 );
this.Controls.Add (this .panel2 );
this.Controls.Add (this .label1 );
this.Controls.Add (this .panel1 );
this.Menu  =  this .mainMenu1 ;
this.Name  =  "WinForm";
this.Text  =  "WinForm";
this.panel1.ResumeLayout ( false );
this.panel2.ResumeLayout ( false );
this.groupBox1.ResumeLayout ( false );
this.tabControl1.ResumeLayout ( false );
this.tabPage1.ResumeLayout ( false );
this.tabPage2.ResumeLayout ( false );
this.ResumeLayout ( false );

// toutes ces lignes correspondent à ceci :
```




```
}
 #endregion
```

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 ){  }
```

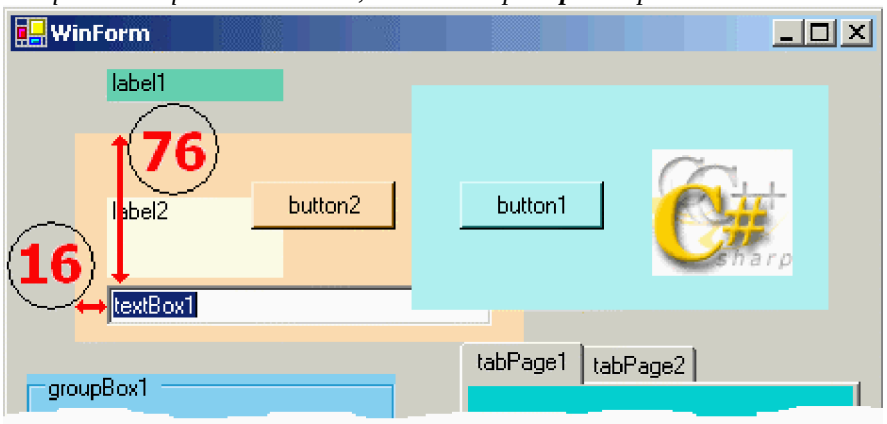
Les lignes d'abonnement sont engendrées dans la méthode InitializeComponent ( ) :

```
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)* :

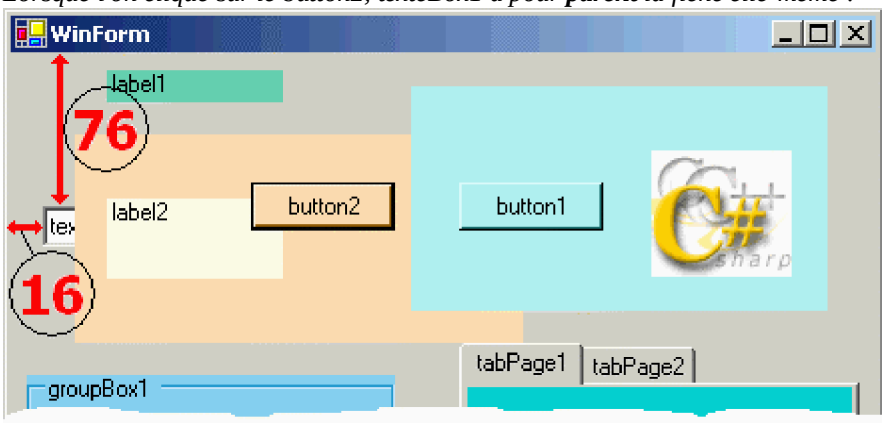
```
// Gestionnaire du click sur button1 :

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

*Lorsque l'on clique sur le button1, texteBox1 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, texteBox1 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 :

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 :

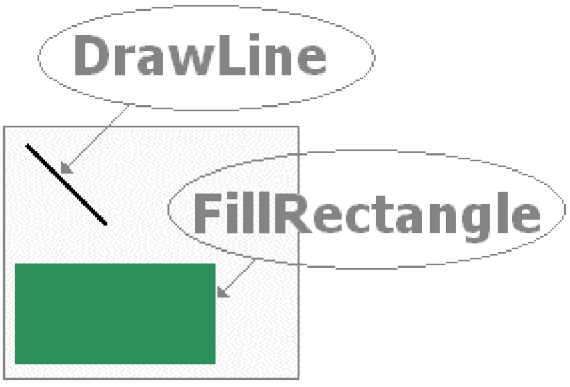
Bibliothèque de classes .NET Framework

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 remplr 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.

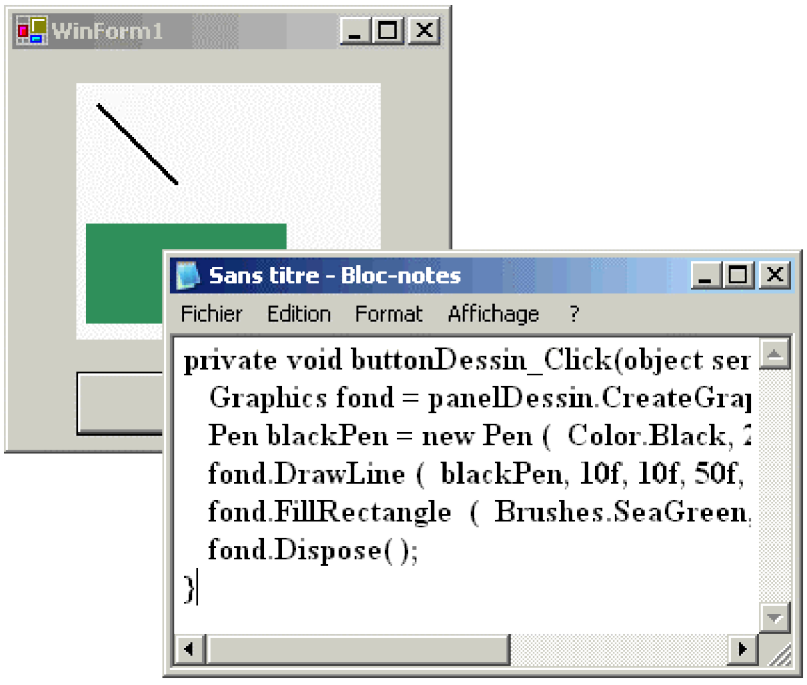
Nous supposons disposer d'un formulaire nommé WinForm1 contenant un panneau nommé panelDessin (**private System.Windows.Forms.Panel** panelDessin) et un bouton nommé buttonDessin (**private System.Windows.Forms.Button** buttonDessin)

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 traçés ;

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

Il existe un moyen simple permettant d'effectuer le redessinement de nos traçés 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( );
}
```

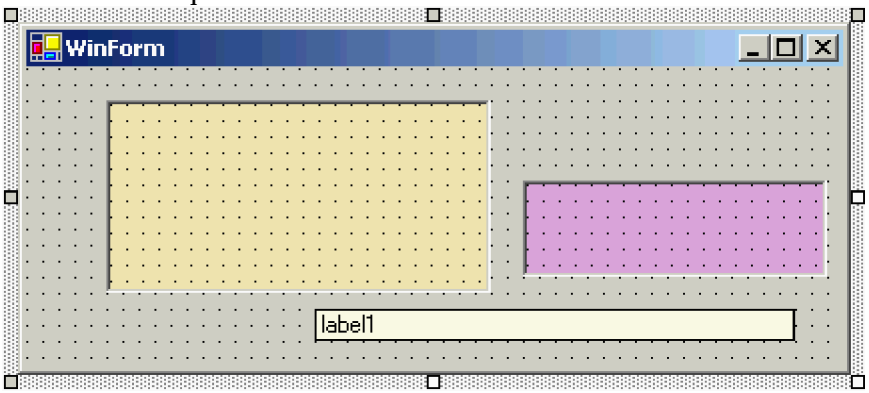
Le RAD a enregistré (abonné) le gestionnaire WinForm1_Paint auprès du délégué Paint dans le corps de la méthode InitializeComponent :

```
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 WinForm et deux panel (panel2 : jaune foncé et panel1 : violet), la label1 ne sert qu'à afficher du texte en sortie :

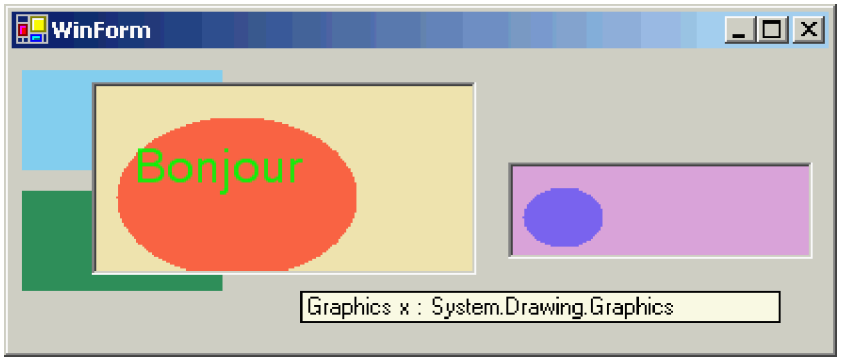


Nous écrivons une méthode **TracerDessin** permettant de dessiner sur le fond de la fiche, sur le fond des deux panel et d'écrire du texte sur le fond d'un panel. La méthode **TracerDessin** est appelée dans le gestionnaire de l'événement Paint du formulaire lors de son redessinement de la fiche afin d'assurer la persistance de tous les traçés. Voici le code que nous créons :

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

private void WinForm_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 );
```

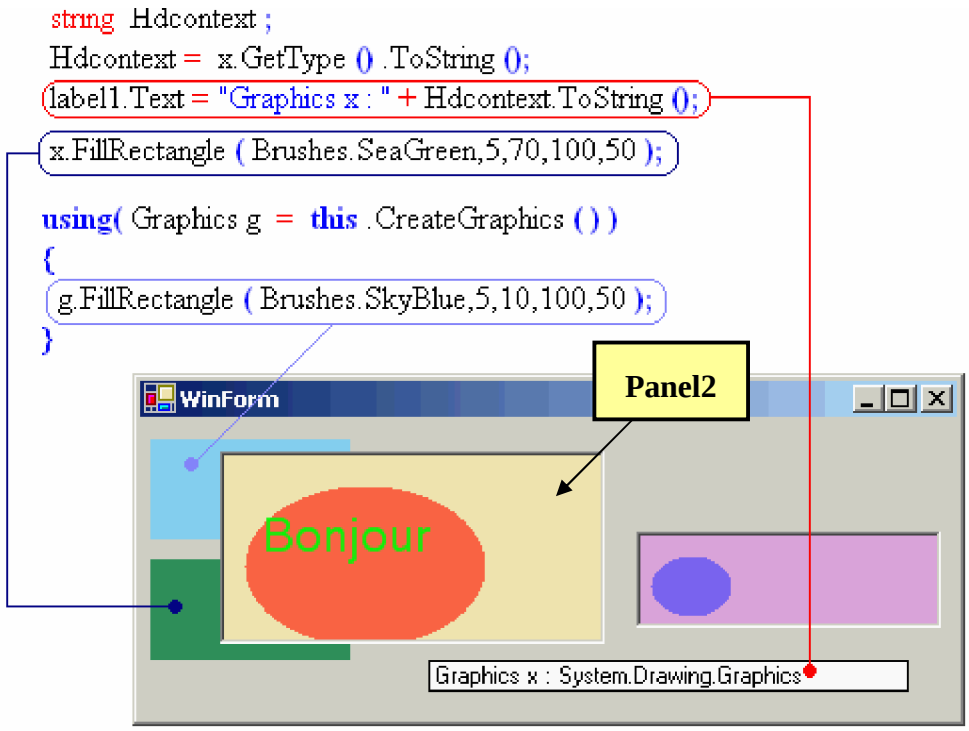
```
 }
}
```

L'exécution de code produit l'affichage suivant :



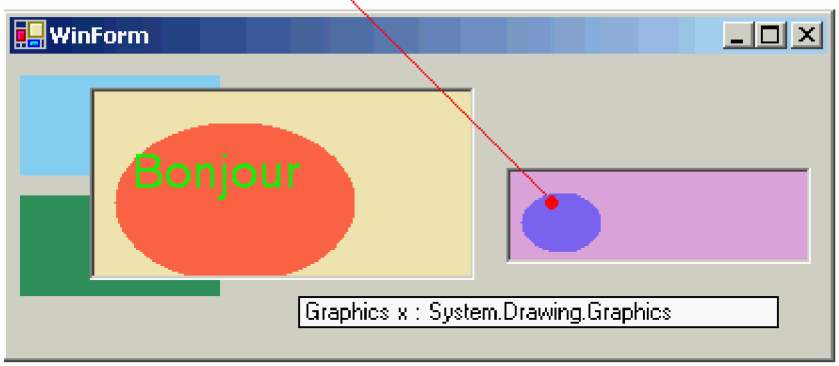
Illustrons les actions de dessin de chaque ligne de code de la méthode TracerDessin :

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



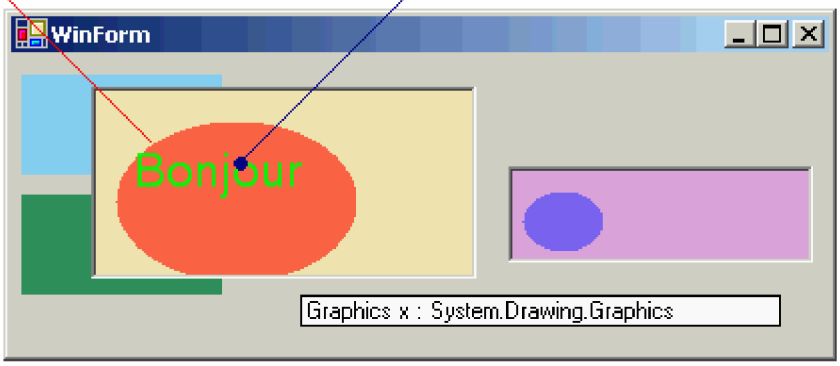
On dessine deux rectangles sur le fond de la fiche, nous notons que ces deux rectangles ont une intersection non vide avec le panel2 (jaune foncé)et que cela n'altère pas le dessin du panel. En effet le panel est un contrôle et donc se redessine lui-même. Nous en déduisons que le fond graphique est situé "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 droite sur le fond du panel1

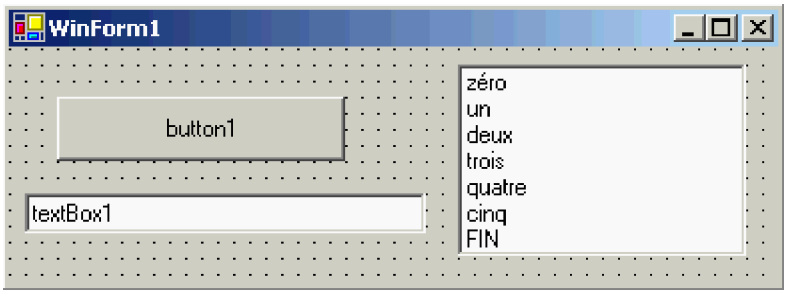
```
using( Graphics h = panel2.CreateGraphics () )
{
h.FillEllipse ( Brushes.Tomato,10,15,120,80 );
h.DrawString ("Bonjour" , new Font (this .Font.FontFamily.Name,18 ) ,Brushes.Lime,15,25 );
}
}
```



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 :



Les graphiques sont gérés dans le code ci-après :

```
namespace ProjWindowsApplication2
{
   /// <summary>
   /// Description Résumé de WinForm1.
   /// </summary>
   public class WinForm1 : System.Windows.Forms.Form
   {
      /// <summary>
      /// Variable requise par le concepteur.
      /// </summary>
      private System.ComponentModel.Container components = null ;
      private System.Windows.Forms.ListBox listBox1;
      private System.Windows.Forms.TextBox textBox1;
      private System.Windows.Forms.Button button1;

      private WinForm1( )
      {
         //
         // Requis pour la gestion du concepteur Windows Form
         //
         InitializeComponent();
         //
         // TODO: Ajouter tout le code du constructeur après l'appel de InitializeComponent
         //
      }
      /// <summary>
      /// Nettoyage des ressources utilisées.
      /// </summary>
      protected override void Dispose (bool disposing)
      {
         if (disposing)
         {
            if (components != null)
            {
               components.Dispose();
            }
         }
         base.Dispose(disposing);
      }

      #region Code généré par le concepteur Windows Form
      /// <summary>
      /// Méthode requise pour la gestion du concepteur - ne pas modifier
      /// le contenu de cette méthode avec l'éditeur de code.
      /// </summary>
      private void InitializeComponent( )
      {
         this.listBox1 = new System.Windows.Forms.ListBox( );
```

```
        this.textBox1 = new System.Windows.Forms.TextBox( );
        this.button1 = new System.Windows.Forms.Button( );
        this.SuspendLayout( );
        //
        // listBox1
        //
        this.listBox1.Items.AddRange ( new object[] {
                "zéro",
                "un",
                "deux",
                "trois",
                "quatre",
                "cinq",
                "FIN" }  );
        this.listBox1.Location = new System.Drawing.Point(224, 8);
        this.listBox1.Name = "listBox1";
        this.listBox1.Size = new System.Drawing.Size(144, 95);
        this.listBox1.TabIndex = 9;
        this.listBox1.SelectedIndexChanged +=
```

```
                                new System.EventHandler ( this.listBox1_SelectedIndexChanged );
        //
        // textBox1
        //
        this.textBox1.Location = new System.Drawing.Point(8, 72);
        this.textBox1.Name = "textBox1";
        this.textBox1.Size = new System.Drawing.Size(200, 20);
        this.textBox1.TabIndex = 8;
        this.textBox1.Text = "textBox1";
        this.textBox1.TextChanged += new System.EventHandler(this.textBox1_TextChanged);
        //
        // button1
        //
        this.button1.Location = new System.Drawing.Point(24, 24);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(144, 32);
        this.button1.TabIndex = 7;
        this.button1.Text = "button1";
        this.button1.Paint += new System.Windows.Forms.PaintEventHandler(this.button1_Paint);
        //
        // WinForm1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.ClientSize = new System.Drawing.Size(384, 117);
        this.Controls.Add(this.listBox1);
        this.Controls.Add(this.textBox1);
        this.Controls.Add(this.button1);
        this.Name = "WinForm1";
        this.StartPosition = System.Windows.Forms.FormStartPosition.Manual;
        this.Text = "WinForm1";
        this.Paint += new System.Windows.Forms.PaintEventHandler(this.WinForm1_Paint);
        this.ResumeLayout(false);
    }
    #endregion

    //-- dessin persistant sur le fond de la fiche par gestionnaire Paint:
    private void WinForm1_Paint(object sender, System.Windows.Forms.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, System.Windows.Forms.PaintEventArgs e)
    {
       Graphics x = e.Graphics;
       x.FillRectangle ( Brushes.SkyBlue,5,5,20,20 );
    }
    //-- dessin non persistant sur le fond du ListBox par gestionnaire SelectedIndexChange :
    private void listBox1_SelectedIndexChanged(object sender, System.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, System.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

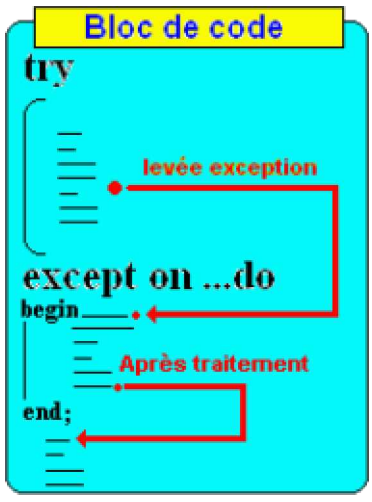
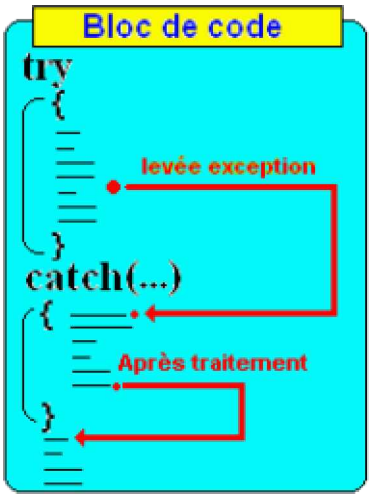
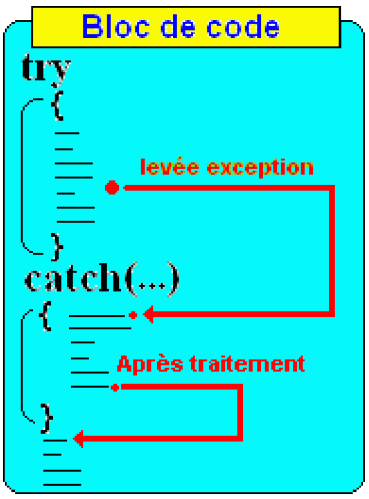
1. similitude et différence

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#

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

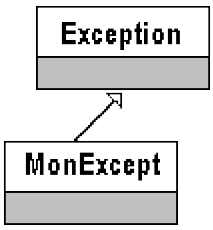
Dans les 3 langages la classes de base des exceptions se nomme : **Exception**

- 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.

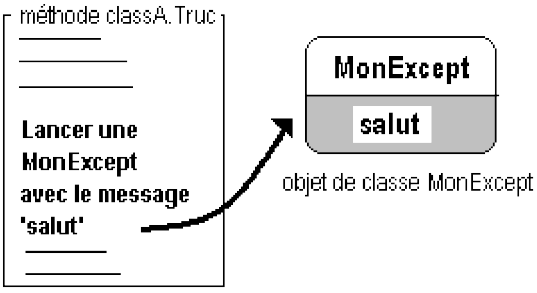
1°) Création d'une nouvelle classe :



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.

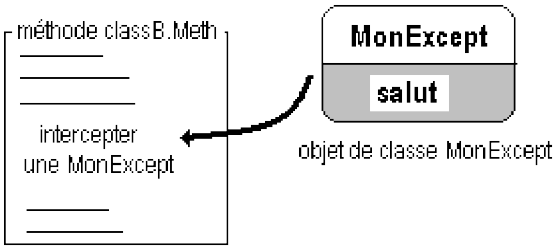
2°) Lancer une MonExcept :



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

Procedure classB.Meth;
begin
  …..
   try
     ….. // code à protéger
   except on MonExcept do
     begin
      ….. // code de réaction à l'exception
     end;
  end;

end;
```

```
class MonExcept extends Exception
{
  public MonExcept (String s)    {
      super(s);
      .........
   }
}

class classB
{
  void Meth ( )  {
      …..
     try
     {
       ….. // code à protéger
     }
     catch (MonExcept e) {
       ….. // code de réaction à l'exception
     }
}
```

```
class MonExcept : Exception
{
  public MonExcept (string s) : base(s)
  {
      .............
   }
}

class classB
{
  void Meth ( )  {
      …..
     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 mémoires 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 accèder 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 contiennet des méthodes d'instances et sont utiles lorsque l'on veut travailler sur un objet de fichier précis.

System.IO.Directory **System.IO.DirectoryInfo**
System.IO.File **System.IO.FileInfo**
System.IO.Path

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

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 et tous ses sous-répertoires 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 et tous ses sous-répertoires 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) **System.IO.Path.GetExtension** (chemin) **System.IO.Path.GetFileName** (chemin) **System.IO.Path.GetFileNameWithoutExtension** (chemin) **System.IO.Path.GetFullPath** ("C:\\rep1\\rep2\\..\\appli.exe") **System.IO.Path.GetPathRoot** (chemin)	C:\rep1\rep2 .exe appli.exe appli C:\rep1\appli.exe 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 **AppendAllText** **AppendText** **Copy** **Create** **CreateText** **Decrypt** **Delete** **Encrypt** **Exists** **GetAccessControl** **GetAttributes** **GetCreationTime** **GetCreationTimeUtc**	**GetLastAccessTime** **GetLastAccessTimeUtc** **GetLastWriteTime** **GetLastWriteTimeUtc** **Move** **Open** **OpenRead** **OpenText** **OpenWrite** **ReadAll** **ReadAllBytes** **ReadAllLines** **ReadAllText** **Replace**	**SetAccessControl** **SetAttributes** **SetCreationTime** **SetCreationTimeUtc** **SetLastAccessTime** **SetLastAccessTimeUtc** **SetLastWriteTime** **SetLastWriteTimeUtc** **WriteAll** **WriteAllBytes** **WriteAllLines** **WriteAllText**

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

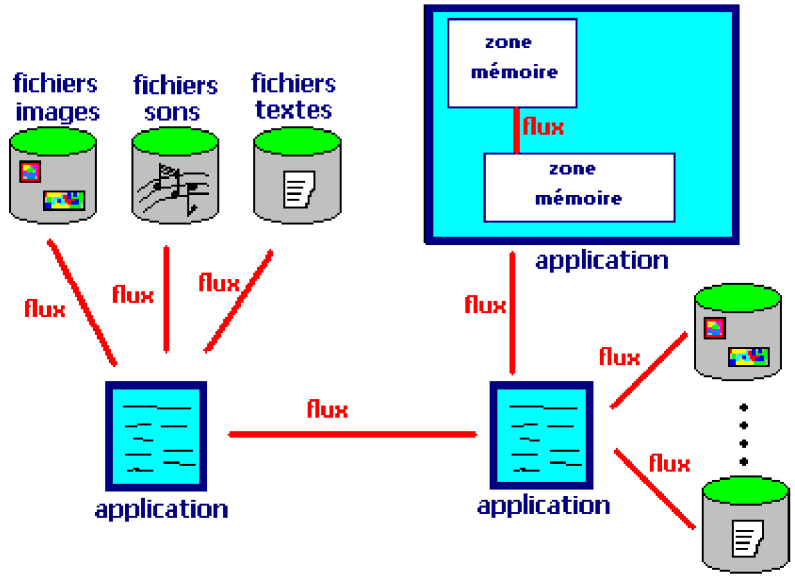
```
string cheminRep = @"d:\temp\rep1\fichier.txt";
// création d'un répertoire et d'un sous-répertoire s'ils n'existent pas déjà
if ( !Directory.Exists( Path.GetDirectoryName(cheminRep ) ) )
      Directory.CreateDirectory( Path.GetDirectoryName(cheminRep ) );

// 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:

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...).

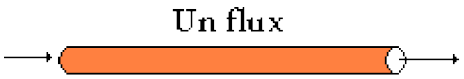
Réciproquement, une application peut après traitement, délivrer en **sortie** des résultats, on dit **écrire**, dans un fichier, vers une autre application ou dans une zone mémoire. **Les flux servent à connecter entre elles ces diverses sources de données**.



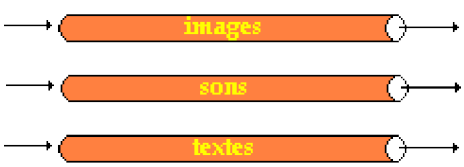
2.1 Les flux avec C#

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'ecriture 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 concrètent** 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

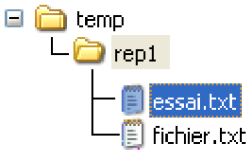
Exemple d'utilisation des 2 classes précédentes.

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.text' 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 >
```

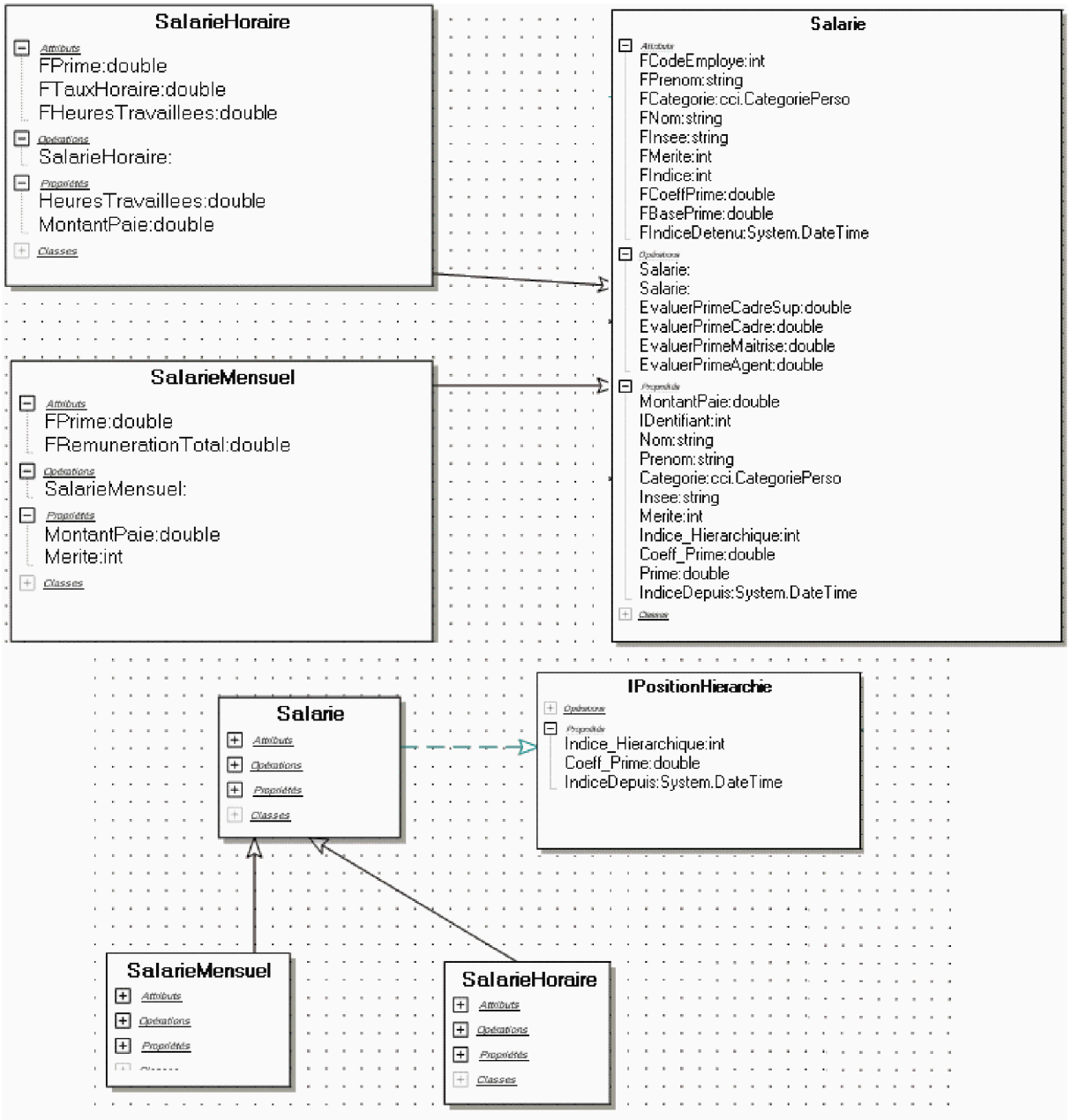
.....
copie < texte stocké par programme ligne N : 9 >

3. Exercice de fichier de salariés avec une IHM

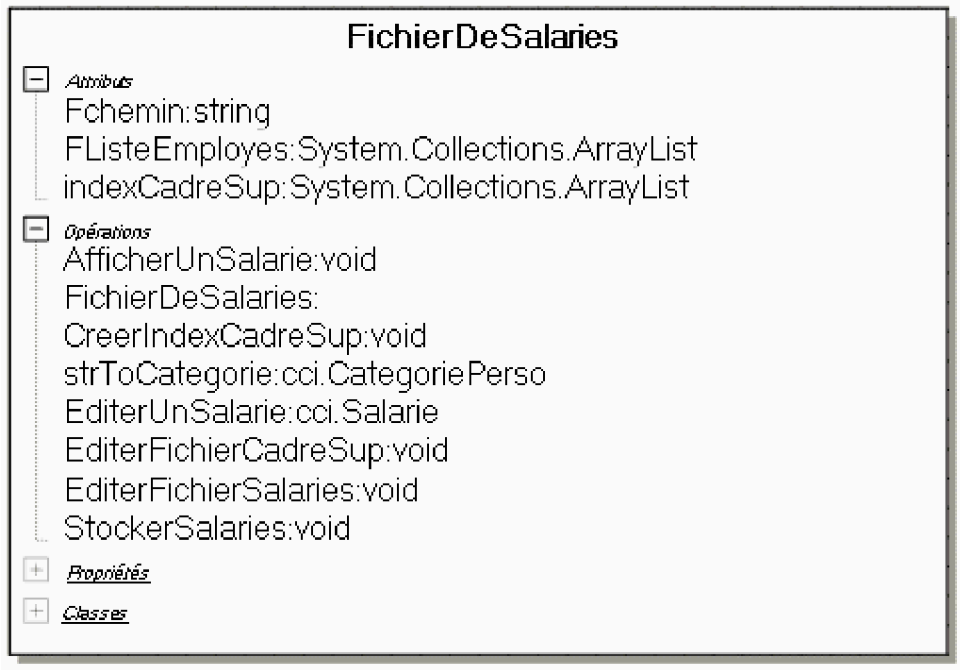
Soit à écrire un programme C# gérant en saisie et en consultation un fichier des salariés d'une petite entreprise.

Ci-dessous les éléments à écrire en mode console d'abord, puis ensuite créez vous-mêmes une interface interactive.

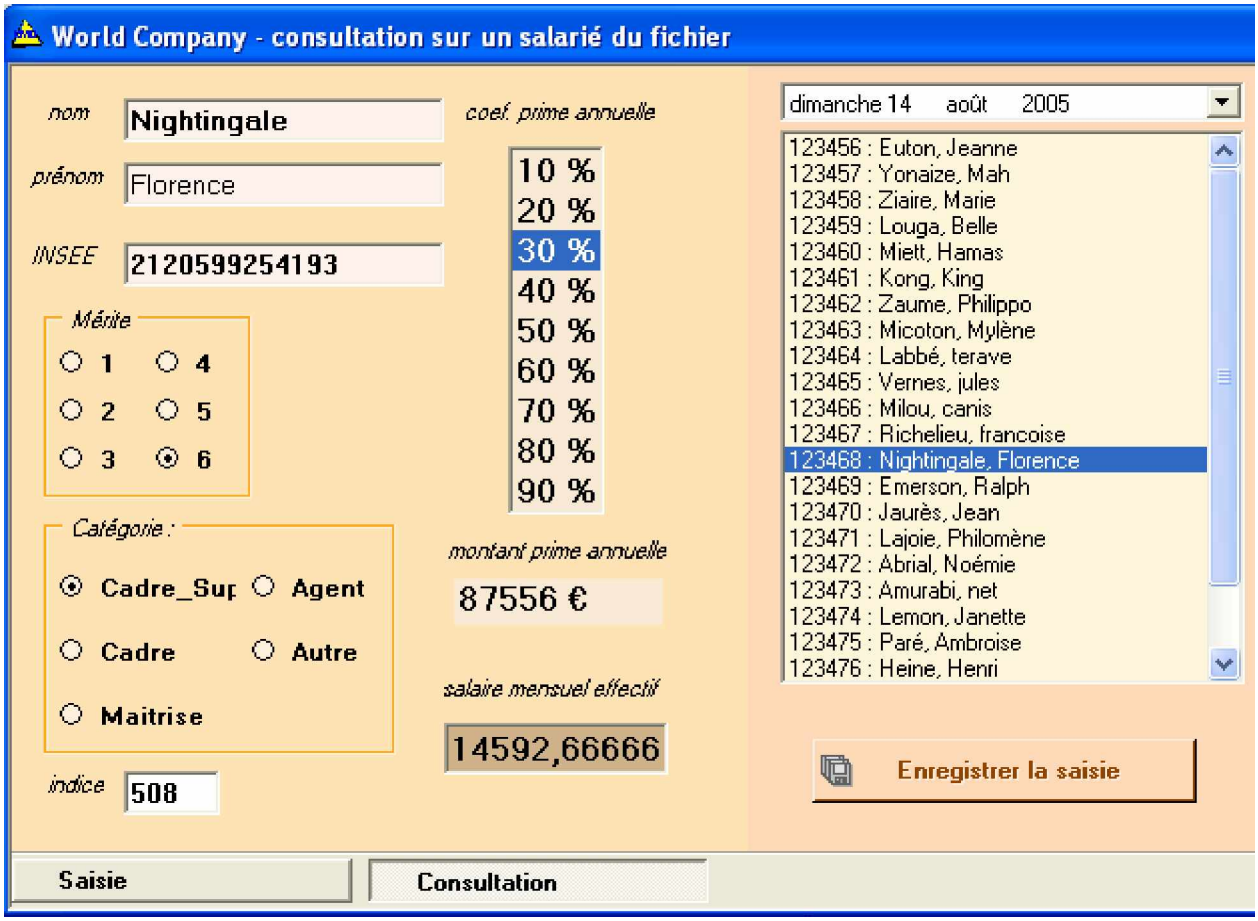
Soit les diagrammes de classe suivants :



Le programme travaille sur un fichier d'objets de classe Salarie :



Un exemple d'IHM possible pour ce programme de gestion de salariés :



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 FCategorie ;
 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 CategoriePerso Categorie {
  get  { return FCategorie ; }
  set  { FCategorie = 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 ( FCategorie )
    {
      case CategoriePerso.Cadre_Sup :  return EvaluerPrimeCadreSup ( FMerite );
      case CategoriePerso.Cadre :  return EvaluerPrimeCadre ( FMerite );
      case CategoriePerso.Maitrise :  return EvaluerPrimeMaitrise ( FMerite );
      case CategoriePerso.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 IDentifiant, string Nom, string Prenom, string Insee, double
TauxHoraire ): base ( IDentifiant, 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 EditerUnSalarie ( int rang ) {
    SalarieMensuel perso ;
     ..........
    perso = new SalarieMensuel ( IDentifiant, 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 Employe1  =  new  SalarieMensuel ( 123456, "Euton" , "Jeanne" ,
     CategoriePerso.Cadre_Sup, "2780258123456" ,6,700,0.5,50000 );
 SalarieMensuel Employe2  =  new  SalarieMensuel ( 123457, "Yonaize" , "Mah" ,
     CategoriePerso.Cadre, "1821113896452" ,5,520,0.42,30000 );
 SalarieMensuel Employe3  =  new  SalarieMensuel ( 123458, "Ziaire" , "Marie" ,
     CategoriePerso.Maitrise, "2801037853781" ,2,678,0.6,20000 );
 SalarieMensuel Employe4  =  new  SalarieMensuel ( 123459, "Louga" , "Belle" ,
     CategoriePerso.Agent, "2790469483167" ,4,805,0.25,20000 );

 ArrayList ListeSalaries  =  new  ArrayList ();
 ListeSalaries.Add (  Employe1  );
 ListeSalaries.Add (  Employe2  );
 ListeSalaries.Add (  Employe3  );
 ListeSalaries.Add (  Employe4  );
 foreach( SalarieMensuel s  in  ListeSalaries )
  InfoSalarie ( s );
 Console .WriteLine (">>>  Promotion indice de " + Employe1.Nom + " dans 2 secondes.");
 Thread.Sleep ( 2000 );
 Employe1.Indice_Hierarchique = 710 ;
 InfoSalarie (  Employe1  );
 //----------------------------------------//
 FichierDeSalaries Fiches  =  new  FichierDeSalaries ("fichierSalaries.txt" ,ListeSalaries );
 Console .WriteLine (">>>  Attente 3 s pour création de nouveaux salariés");
 Thread.Sleep ( 3000 );
 Employe1  =  new  SalarieMensuel ( 123460, "Miett" , "Hamas" ,
     CategoriePerso.Cadre_Sup, "1750258123456" ,4,500,0.7,42000 );
 Employe2  =  new  SalarieMensuel ( 123461, "Kong" , "King" ,
     CategoriePerso.Cadre, "1640517896452" ,4,305,0.62,28000 );
 Employe3  =  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
16914,1666666667
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
Louga
Belle
```

*Agent
2790469483167
4
805
15/02/2004 19:52:36
0,25
7010
2250,83333333333
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

Résultats console :

```
Employé n&deg;123456: Euton / Jeanne
 n&deg; 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&deg;123457: Yonaize / Mah
 n&deg; 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: 7266,66666666667
   coeff prime : 0,5
   montant prime annuelle : 67600
   montant paie mensuelle: 8133,33333333333
   coeff prime : 0,6
   montant prime annuelle : 80600
   montant paie mensuelle: 9216,66666666667
   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: 6216,66666666667
   coeff mérite : 1
   montant prime annuelle : 47120
   montant paie mensuelle: 6426,66666666667
   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&deg;123459: Louga / Belle
 n&deg; 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: 3420,83333333333
  coeff prime : 1
  montant prime annuelle : 23210
  montant paie mensuelle: 3600,83333333333
  ----------------------
```

```
  coeff mérite : 0
  montant prime annuelle : 6610
  montant paie mensuelle: 2217,5
  coeff mérite : 1
  montant prime annuelle : 6710
  montant paie mensuelle: 2225,83333333333
  coeff mérite : 2
  montant prime annuelle : 6810
  montant paie mensuelle: 2234,16666666667
  coeff mérite : 3
  montant prime annuelle : 6910
  montant paie mensuelle: 2242,5
  coeff mérite : 4
  montant prime annuelle : 7010
  montant paie mensuelle: 2250,83333333333
  coeff mérite : 5
  montant prime annuelle : 7110
  montant paie mensuelle: 2259,16666666667
  coeff mérite : 6
  montant prime annuelle : 7210
  montant paie mensuelle: 2267,5
  coeff mérite : 7
  montant prime annuelle : 7310
  montant paie mensuelle: 2275,83333333333
  coeff mérite : 8
  montant prime annuelle : 7410
  montant paie mensuelle: 2284,16666666667
  coeff mérite : 9
  montant prime annuelle : 7510
  montant paie mensuelle: 2292,5
=====================================
>>>  Promotion indice de Euton dans 2 secondes.
Employé n&deg;123456: Euton / Jeanne
 n&deg; 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: 16914,1666666667
  coeff prime : 0,5
  montant prime annuelle : 152970
  montant paie mensuelle: 16914,1666666667
  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
7266,66666666667
123458
Ziaire
Marie
*Maitrise
2801037853781
2
678
15/02/2004 19:52:36
0,6
34434
4536,16666666667
123459
Louga
Belle
*Agent
2790469483167
4
805
15/02/2004 19:52:36
0,25
7010
2250,83333333333
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° SS : 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: Miett / Hamas
n° SS : 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;`