



Compléments C# / .NET

EI₃ SAGI

*ISTIA
2017-2018*

1. Compléments (Applications à fenêtres, Sérialisation, GDI+)

Le framework .NET est extrêmement riche (le livre C# in a Nutshell fait 1000 pages, pourtant il ne fait qu'introduire les classes .Net et n'aborde même pas la programmation d'applications fenêtrées dans Windows). .Net fournit une librairie de classes pour la gestion de données, la communication, l'affichage ... Ce chapitre regroupe l'introduction de quelques documents apportant un complément au langage C#.

Notamment, on décrit des techniques liées à l'écriture d'applications à fenêtres dans l'OS Windows. (Applications WinForm). Cette partie permet de découvrir les notions essentielles de ce type d'applications, notamment la place importante des événements.

Ensuite, sont donnés quelques points d'entrée sur la sérialisation binaire (qui permet de sauvegarder un objet sur fichier binaire) et la gestion de l'affichage graphique (GDI+) dans les application fenêtrées.

Documents Complémentaires

Applications Windows C#/.NET (Résumé)

Structure

Formulaires et boîtes de dialogue

Application MDI

Expressions Régulières (traitement de chaînes)

Flux I/O (dont les fichiers)

Communication réseau (API Socket)

Introduction à la sérialisation avec .NET

Introduction à GDI+ (Graphical Device Interface)

Applications Windows C#/.NET (Résumé)

Ce document fournit des indications sur la création d'applications Windows (avec fenêtres) en C# pour .NET. En revanche, peu d'indications sont données sur l'exploitation des contrôles (pour cela, se référer à la documentation en ligne (msdn.microsoft.com)).

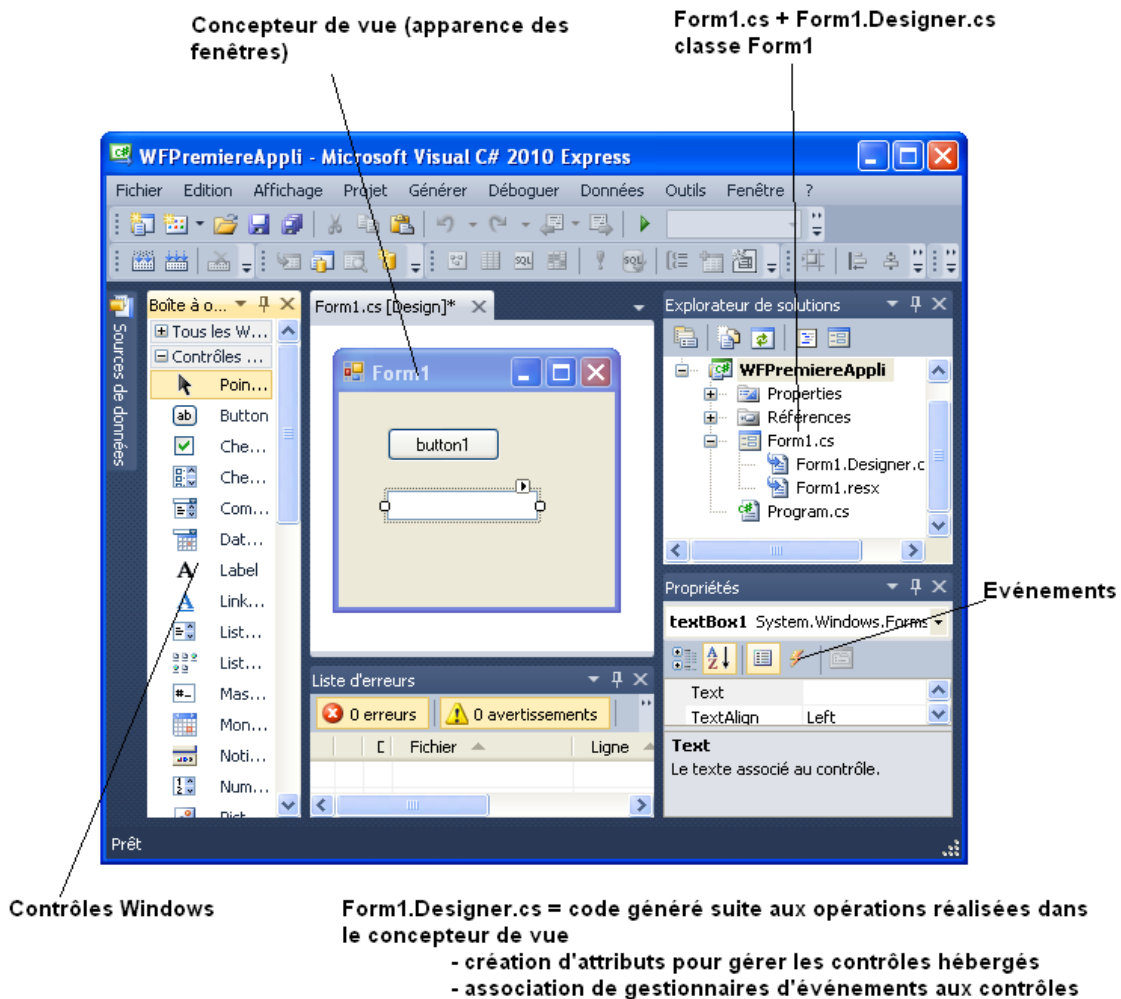
Pour les applications Windows C#/.NET, on peut utiliser Visual Studio ou la version Visual C# Express. Dans ces IDE, la création du projet est effectuée par :

Fichier / Nouveau Projet / Visual C# / Application Windows Forms
Nom :
Emplacement :

Important : Le nom de projet sert également de nom de **namespace**.

1.1. Structure d'une application Windows

Suite à la création du projet, les différentes fenêtres de l'IDE vous permettent d'atteindre du code (explorateur de solution), de modifier l'apparence des fenêtres (concepteur de vue), de visualiser les propriétés et événements des contrôles.



Exemple : application avec deux contrôles Button + TextBox

Form1 = classe de la fenêtre principale

Une instance de cette classe est générée dans la fonction Main() (Program.cs)

Form1 est écrite sur 2 fichiers (mot clé **partial**) (**Form1.cs + Form1.Designer.cs**)

Form1.cs : fichier modifiable par nos soins (gestionnaires d'événements)

Form1.Designer.cs : modifié par l'IDE suite aux opérations faites dans le concepteur de vue.

QUAND ON AJOUTE UN CONTRÔLE :

- attribut supplémentaire dans la classe **Form1** (voir Form1.Designer.cs)
- instantiation de la classe du contrôle dans la méthode **Form1.InitializeComponent()**;

//Form1.cs (Une partie de la classe Form1) PEUT-ETRE EDITE

```
namespace WFPremiereAppli
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent(); // Cette méthode crée les objets Contrôles
                                   // Voir fichier Form1.Designer.cs
        }
    }
}
```

//Form1.Designer.cs (Autre partie de Form1) MIS A JOUR PAR Visual Studio

```
namespace WFPremiereAppli
{
    partial class Form1
    {
        ...

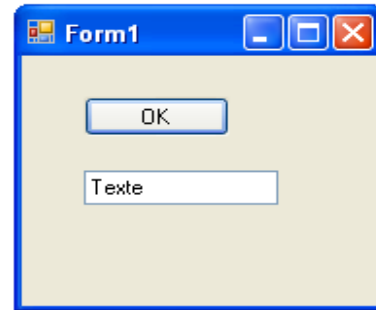
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button(); //instanciation
            this.textBox1 = new System.Windows.Forms.TextBox();
            ...
            this.button1.Text = "button1";
            this.button1.UseVisualStyleBackColor = true;
            this.textBox1.Name = "textBox1";
            ...
            this.Controls.Add(this.textBox1);
            this.Controls.Add(this.button1);
            ...
        }

        private System.Windows.Forms.Button button1; // attribut de type Button
        private System.Windows.Forms.TextBox textBox1; // attribut de type TextBox
    }
}
```

1.2. Accès aux contrôles dans le code

Depuis n'importe quelle méthode de la classe Form1, les contrôles hébergés par la fenêtre principale sont atteignables grâce à leurs noms d'attributs.

```
namespace WFPremiereAppli
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            button1.Text = "OK";
            textBox1.Text = "Texte";
        }
    }
}
```



Quelques propriétés communes aux contrôles

Text = (type string) définit le texte d'un label, d'un textbox, d'un bouton ...

Location = (type Point) position X,Y du contrôle [(0,0) en haut à gauche]

Size = (type Size) taille X,Y du contrôle

Enabled = (type bool) activation du contrôle

Tag = (type object) n'importe quelle variable/objet que l'on "attache" au contrôle

```
public partial class Form1 : Form
{
    public Form1(){
        InitializeComponent();
        button1.Text = "OK";
        button1.Location = new System.Drawing.Point(0, 0);
        button1.Size = new System.Drawing.Size(100, 20);
        button1.Enabled = true;
        button1.Tag = 1;
        textBox1.Text = "Texte";
        textBox1.Enabled = false;
        textBox1.Tag = 2;
    }
}
```



1.3. Gestionnaires d'événements

Les contrôles déclenchent des événements auxquels des méthodes (gestionnaires d'événements) peuvent s'abonner. Pour associer un gestionnaire d'événements à l'événement d'un contrôle :

- 1) sélectionner le contrôle (concepteur de vue)
- 2) Fenêtre Propriétés | Onglet  → Double Click sur l'événement

Exemple : Gestionnaire de l'événement **Click** d'un bouton

```
// Form1.cs
public partial class Form1 : Form{
    public Form1(){
        InitializeComponent();
    }

    private void button1_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Appui sur le bouton");
    }
}
```

```
// Form1.Designer.cs
namespace WFPremiereAppli
{
    partial class Form1
    {
        ...

        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();    //instanciation
            ...
            this.button1.Click += new System.EventHandler(this.button1_Click);
        }

        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.TextBox textBox1;
    }
}
```



NOTE : Pour toutes ces notions, il est recommandé de revoir la section événement/delegate.

Signature des événements (la délégation associée)

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

Les événements peuvent donc déclencher des appels de méthodes avec deux paramètres

sender = référence à l'objet ayant déclenché l'événement (souvent un contrôle)

e = arguments (données) fournies au gestionnaire d'événement

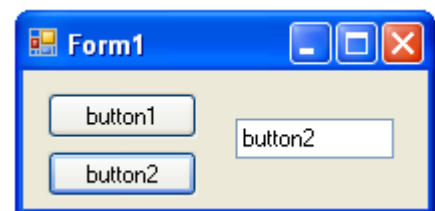
1.4. Même gestionnaire pour plusieurs événements

On peut associer le même gestionnaire aux événements de plusieurs contrôles.

- 1) sélectionner le deuxième contrôle (concepteur de vue)
- 2) Fenêtre Propriétés | Onglet  → choisir un gestionnaire existant dans la liste

```
public partial class Form1 : Form
{
    public Form1(){    InitializeComponent(); }

    // même gestionnaire click pour deux boutons
    private void button1_Click(object sender, EventArgs e)
    {
        Button b = sender as Button; // conversion object vers Button
        if (b != null) // si l'émetteur est réellement un Button
        {
            textBox1.Text = b.Name;
        }
    }
}
```



```
// Form1.Designer.cs
namespace WFPremiereAppli
{
    partial class Form1
    {
        ...

        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button(); //instanciation
            this.button2 = new System.Windows.Forms.Button(); //instanciation
            ...
            this.button1.Click += new System.EventHandler(this.button1_Click);
            this.button2.Click += new System.EventHandler(this.button1_Click);
        }
        private System.Windows.Forms.Button button1;
        private System.Windows.Forms.Button button2;
        private System.Windows.Forms.TextBox textBox1;
    }
}
```

↑
Même gestionnaire
pour les
événements de 2
contrôles différents

1.5. Formulaire (Fenêtre) Supplémentaire

Un formulaire principal est ouvert avec l'application (Form1 par défaut). On peut avoir d'autres formulaires dans l'application : des boîtes de message (formulaires clé en main), des formulaires et boîtes de dialogue.

Classe MessageBox Crée une boîte de dialogue pour afficher un message, ou poser une question à l'utilisateur.

```
private void buttonFormulaire_Click(object sender, EventArgs e)
{
    MessageBox.Show("Message seul");
    MessageBox.Show("Message", "Titre de la boîte");

    // MessageBoxButtons = énumération de différentes configurations
    DialogResult result;
    result = MessageBox.Show("Continuer ?", "Attention", MessageBoxButtons.OKCancel);
    if (result == DialogResult.OK)
    {
        MessageBox.Show("Vous avez cliqué su OK");
    }
}
```

Formulaire

Pour ajouter une nouvelle classe de formulaire : dans l'explorateur de solutions, click-droit sur nom de projet puis Ajouter/Formulaire Windows.

Dans l'explorateur de solution, on voit alors deux nouveaux fichiers pour le formulaire supplémentaire.

Note : La classe du formulaire supplémentaire dérive de la classe **Form**.

Ex : FormSupp.cs + FormSupp.Designer.cs

Ce nouveau formulaire peut lui-même contenir des contrôles qui seront vus comme des attributs de la classe, et des gestionnaires associés aux contrôles.

OUVERTURE MODALE (formulaire bloquant)

On n'a plus accès au reste de l'application avant la fermeture du formulaire.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }
    private void buttonFormulaire_Click(object sender, EventArgs e)
    {
        FormSupp form = new FormSupp();

        form.ShowDialog();    // FORMULAIRE MODAL (Genre Boîte de Dialogue)
    }
}
```

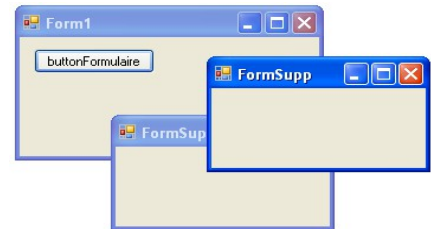


OUVERTURE NON MODALE

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void buttonFormulaire_Click(object sender, EventArgs e)
    {
        FormSupp form = new FormSupp();

        form.Show();    // FORMULAIRE NON MODAL
    }
}
```



Note : le problème principal est, dans ce cas, l'échange de données entre les fenêtres.

1.6. Boîte de Dialogue (Formulaire Modal)

Une boîte de dialogue sert principalement à demander à l'utilisateur des données. La version la plus courante est donc constituée de deux boutons (OK/Cancel) et des contrôles permettant des entrées utilisateur.

A noter : les boîtes .NET (**OpenFileDialog**, **ColorDialog** ...) reprennent ce principe d'échange.

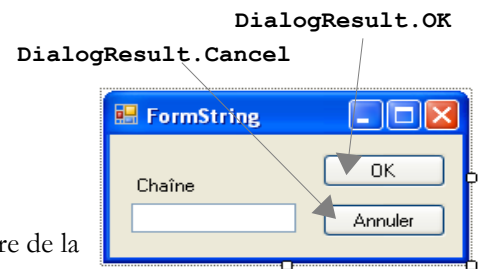
Exemple : *boîte de dialogue pour demander une chaîne de caractères à l'utilisateur*

- Ajouter formulaire (classe **FormString**)
(explorateur de solution/click droit/ ajouter *Formulaire W*.)
- Ajouter deux contrôles button (OK/Annuler)
- Ajouter les contrôles pour entrées/sorties utilisateur (textbox, labels ...)

IMPORTANT : définir la propriété **DialogResult** des boutons Ok/Annuler aux valeurs suivantes

DialogResult.OK pour le bouton OK
DialogResult.Cancel pour le bouton Cancel

Dès lors, ces boutons vont fermer la boîte en associant une valeur à la fermeture. Cette donnée sera exploitée comme valeur de retour à la fermeture de la boîte modale.



La boîte de dialogue est affichée avec la méthode **ShowDialog()** qui est bloquante (tant que le formulaire est ouvert) et fournit en retour un **DialogResult**, c'est-à-dire le bouton utilisé pour la fermeture du formulaire.

PRINCIPE D'ECHANGE COTE UTILISATEUR DE LA BOÎTE

La boîte gère ses contrôles. On ne fournit à/ne récupère de la boîte que les données qu'elle doit afficher/ fournir. On peut, par exemple, définir une propriété dans la classe du formulaire pour les échanges de données.

Exemple : la boîte **FormString** échange des données à travers une propriété **Chaîne** (string)

```
FormString dlg = new FormString(); // on crée un objet

// facultatif : on passe des données à la boîte (avant ouverture)
dlg.Chaîne = "Ma Chaîne";
if (dlg.ShowDialog() == DialogResult.OK)
{
    // si l'utilisateur ferme avec OK
    _____ = dlg.Chaîne; // récupère la chaîne
}
```

Echanges de données : le code utilisateur fournit au formulaire modal les données à afficher, et récupère (après fermeture) les données saisies dans la boîte. Côté utilisateur de la boîte, on n'a pas besoin de connaître ou gérer les contrôles de la boîte. On n'exploite que les données (souvent des propriétés) affichées par l'objet **Form_____**.

PRINCIPE D'ECHANGE COTE BOÎTE DE DIALOGUE

Pour que la boîte de dialogue puisse s'utiliser ainsi, il va falloir compléter la classe de la boîte de dialogue (**FormString**) et programmer certains événements (ouverture, demande de fermeture)

-A l'ouverture de la boîte (événement **Load**) : il faut initialiser les contrôles hébergés à partir des données fournies par le code utilisateur (cf. paragraphe précédent). Ici, le code utilisateur a renseigné la propriété **Chaîne** avant l'ouverture. Donc côté boîte, cette propriété est supposée déjà renseignée.

- A la demande de fermeture de la boîte (événement **FormClosing**) : il faut faire l'inverse. On renseigne la propriété **Chaîne** (qui sera exploitée par le code utilisateur de la boîte)

```
public partial class FormString : Form
{
    // attribut pour la chaîne
    private string _str;

    // prop. Chaîne (vue par le code utilisateur)
    public string Chaîne{
        get { return _str; }
        set { _str = value; }
    }
}
```

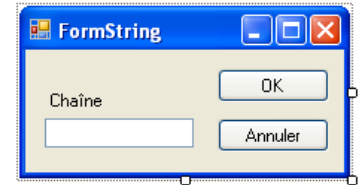
```

public FormString() {
    InitializeComponent(); // création des contrôles hébergés
}

// Gestionnaire event Load
private void FormString_Load(object sender,
                             EventArgs e)
{
    textBoxStr.Text = Chaîne;
}

// Gestionnaire event FormClosing
private void FormString_FormClosing(object sender, FormClosingEventArgs e)
{
    Chaîne = textBoxStr.Text;
}
}

```



EMPECHER LA FERMETURE DE LA BOÎTE (ex : problème de format)

L'événement **FormClosing** est déclenché sur une *demande* de fermeture. Il est alors encore possible d'annuler la fermeture. Ceci permet de vérifier si les données fournies par l'utilisateur ont un format valide.

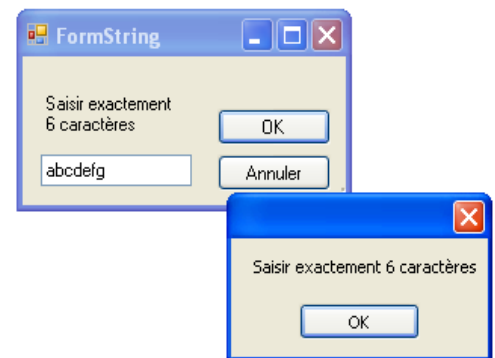
Exemple : imposer que 6 caractères soient saisis

```

public partial class FormString : Form
{
    private string _str;
    public string Chaîne
    {
        get { return _str; }
        set { _str = value; }
    }
    ...
    private void FormString_FormClosing(object sender, FormClosingEventArgs e)
    {
        if (DialogResult == DialogResult.OK)
        {
            if (textBoxStr.Text.Length != 6)
            {
                MessageBox.Show("Saisir exactement 6 caractères");

                e.Cancel = true; //Annule (la boîte reste ouverte donc)
            }
            else Chaîne = textBoxStr.Text;
        }
    }
}

```



1.7. Application MDI

Les applications MDI (Multi Document Interface) sont celles qui permettent d'agir simultanément sur plusieurs documents par l'intermédiaire de plusieurs fenêtres. On va se contenter ici de gérer un seul jeu de données (un document) mais plusieurs vues (plusieurs formulaires) pour afficher les données.

Pour avoir une application MDI (avec plusieurs fenêtres clientes de la fenêtre principale), il faut que la classe de formulaire principale **Form1** ait sa propriété **IsMdiContainer** à **True**. Dès lors, la zone cliente de la fenêtre sert à héberger d'autres formulaires Windows.



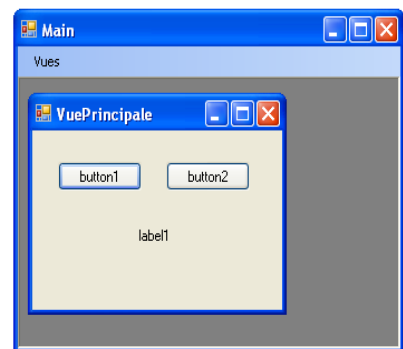
Formulaire client de la fenêtre principale

Ajouter un formulaire (classe **VuePrincipale**). Sur l'événement **Load** de la forme **Form1** créer un formulaire pour la vue cliente de la façon suivante :

```
public partial class Form1 : Form
{
    public Form1 ()
    {
        InitializeComponent();
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        VuePrincipale vueP = new VuePrincipale();
        vueP.MdiParent = this;
        // la vue cliente a la forme principale comme parent MDI

        vueP.Show(); // formulaire non modal
    }
}
```



Application MDI ayant une structure Document/Vues

Design Pattern Singleton pour que le Document soit accessible facilement

Lorsqu'il y a plusieurs fenêtres dans une même application, elles ont généralement besoin de données communes. On peut stocker les données communes dans un objet **Document** accessible par toutes les fenêtres.

```
public class Document
{
    public static readonly Document doc = new Document();
    private Document() { }
}
```

De cette façon, l'objet de la classe **Document** est unique et est accessible depuis partout par la syntaxe

Document.doc

Utilisation des événements pour synchroniser les fenêtres (design pattern Observer)

Un problème fréquent dans les applications MDI est d'assurer la synchronisation de l'affichage avec les changements de données du document : on modifie une donnée via une fenêtre et c'est une autre fenêtre qui doit être remise à jour.

Pour que toutes les fenêtres soient synchronisées sur les évolutions des données, et qu'elles soient donc informées des changements de données (qui peuvent provenir d'actions dans d'autres fenêtres), il est utile que la classe **Document** propose un *événement* auquel les vues peuvent s'abonner.

```
public class Document
{
    delegate ... DocEventHandler( ... ); // signature de l'événement
    event DocEventHandler DocChanged; // événement

    public static readonly Document doc = new Document();
    private Document() { }
}
```

L'objet **Document** peut alors déclencher son événement **DocChanged** pour signaler une évolution des données.

Pour être informées, il suffit que les fenêtres abonnent un gestionnaire à cet événement.

Note : ceci correspond à l'application du design pattern *Observer* dans .Net (cf. Cours Génie Logiciel)

Méthode du formulaire abonnée à l'événement **Document.doc.DocChanged**

Les fenêtres clientes MDI doivent donc abonner une méthode à l'événement **DocChanged** du document et attendre que le document les prévienne de ses changements d'état.

```
public partial class VuePrincipale : Form
{
    public VuePrincipale()
    {
        InitializeComponent();
        Document.doc.DocChanged += new DocEventHandler (OnDocChanged);
    }

    private void OnDocChanged( ... ) // signature DocEventHandler
    {
        // gestion de la Mise à Jour de la vue
        ...
    }
}
```

1.8. Les Expressions Régulières (classe Regex)

Mécanisme fourni par .NET pour vérifier qu'une suite de caractères (une chaîne) satisfait un "motif" (pattern) décrit par une expression régulière, c'est-à-dire une expression finie pouvant néanmoins décrire des répétitions et des choix. Grâce à ce mécanisme on peut vérifier qu'une chaîne de caractères satisfait un format imposé.

Note : les types numériques (int,float,decimal) fournissent déjà des méthodes Parse/TryParse pour vérifier le format et convertir une chaîne en valeur si le format est correct.

Les expressions régulières permettent de vérifier qu'une suite de caractères satisfait certaines contraintes : exemples :

est-ce une suite de caractères tous compris entre '0' et '9' (format valable pour un entier base 10) ?
 "123" -> OK "1e4" -> KO
 est-ce une suite de caractères, sans chiffre, d'au moins 2 lettres et au plus 5 lettres ?
 "art" -> OK "rtaert" -> KO (6 lettres) "ra4te" -> KO (pas uniquement des lettres)
 est-ce une suite de caractères sans espace commençant par une lettre ?
 "1ef23" -> KO "zy4t" -> OK
 est-ce une suite de caractères où apparaissent 3 sous-chaînes avec la structure "entier;" ?
 "12;3;4;" -> OK "12;3;4;6;" -> KO

Les objets **Regex** vont permettre de décrire par une expression finie (pattern) ce qu'on vient de décrire littéralement par des phrases.

Les expressions régulières .NET utilisent une chaîne de caractères pour décrire le motif (pattern). Ce motif représente le format attendu au moyen de caractères spéciaux. Ensuite, ce motif peut être appliqué à différentes chaînes de caractères pour vérifier si le format est correct, voire même si plusieurs sous-chaînes satisfont le format.

```
// Y-a-t-il 2 caractères successifs dans l'intervalle 'a' - 'g' ?
Regex re = new Regex(@"[a-g]{2}"); // motif "[a-g]{2}"
Match m = re.Match("chaîne de caracteres"); // chaîne testée
if (m.Success) // s'il y a une concordance (match)
{
    Console.WriteLine(m.Value); // = "de" (sous-chaîne trouvée)
    Console.WriteLine(m.Index); // = 7 (position dans la chaîne)
    Console.WriteLine(m.Length); // = 2 (taille de la sous-chaîne)
}
```

Note : si plusieurs sous-chaînes concordent avec le motif, la concordance correspond par défaut à la première dans le sens d'exploration gauche->droite.

Pour trouver toutes les sous-chaînes qui concordent avec le motif :

```
Regex re = new Regex(@"[a-g]{2}");
MatchCollection mc = re.Matches("chaîne de caracteres");
foreach(Match m in mc) // pour chaque concordance
{
    Console.WriteLine(m.Value);
    Console.WriteLine(m.Index);
    Console.WriteLine(m.Length);
}
```

Pour cet exemple, dans "chaîne de caracteres" on a le motif "[a-g]{2}"
 "de" en position 7, "ca" en position 10 et "ac" en position 13

Le langage des patterns

Pour l'essentiel, l'utilisation des expressions régulières requiert de comprendre la construction des patterns. La chaîne de caractères du pattern est constituée de caractères "classiques" et de commandes spécifiques qui ont un sens particulier dans le langage des patterns. On fournit ci-dessous les différents éléments de construction des patterns.

(1) : les caractères `.$^{\{ [ ( | ) * + ? \ \}` ont un sens particulier dans les patterns

Commande	Type de concordance	Pattern	Chaîne et résultat
Tous les caractères en dehors de (1)	Ces caractères n'ont pas d'autre sens et testent la concordance avec eux-même	" ta "	"ta" en positions 0 et 5 dans "ta patate"
\t	Teste la concordance avec une tabulation		
\r	Teste la concordance avec retour chariot		
\f	Teste la concordance avec form feed		
\n	Teste la concordance avec new line		
\xHH	Teste la concordance avec un caractère ASCII dont le code hexa à 2 digits est HH	" \x61 "	"a" en positions 0,3,5,7,10 dans "abracadabra"
\uHHHH	Teste la concordance avec un caractère UTF-16 dont le code hexa à 4 digits est HHHH	" \u00E7 "	"ç" en position 2 dans "leçon"
\\. \\\$ \\{ \\[\\(\\ \\) * \\+ \\? \\\\"	Teste la concordance avec les caractères spéciaux (1) .,\$...	" \\{1\\} "	"{1}" en position 2 dans "a={1}"

Ensemble de caractères

[character_group]	Teste la concordance avec un caractère du groupe. Sensible à la casse par défaut.	"[a-e]"	"a" en position 3 dans "gray" "a","e" dans "lane"
[^ character_group]	Négation : teste la concordance avec les caractères non présents dans le groupe	"[^aei]"	"r","g","n" dans "reign"
[first - last]	Character range: teste la concordance avec l'un des caractères de l'intervalle	"[A-Z]"	"A","B" dans "AB123"
.	Wildcard: teste la concordance avec n'importe quel caractère excepté \n.	"a.e"	"ave" dans "nave" "ate" dans "water"
\w	Teste la concordance avec n'importe quel "word character." (lettre + _)	"\w"	"I","D","A","1", "3" dans "ID A1.3"
\W	Négation de \w	"\W"	" ", "." dans "ID A1.3"
\p{ name }	Teste la concordance avec un caractère de la catégorie Unicode 'name'(voir les catégories Unicode).Ex : Lu (Letter, Uppercase), Ll(Letter, Lowercase), Nd (Number, decimal digit) ...	"\p{Lu}" "p\{Ll}"	"Ä","Q" dans "Äbd Qè" "b","d","è" dans "Äbd Qè"
\P{name}	Négation de \p{name}		
\s	Teste la concordance avec un white-space character (espace, tab, ...)	"\w\s"	"D " dans "ID A1.3"

\S	Négation de \s	"\s\S"	"_" dans "int_ctr"
\d	Digit décimal	"\d\d"	"42" dans "a=423"
\D	Négation de \d : caractère non digit décimal	"\D"	"a=" dans "a=423"

Quantificateurs : commandes et caractères spéciaux qui décrivent les motifs répétés

*	L'élément précédent 0 ou plusieurs fois (greedy : prend le maximum d'occurrences)	"\d*\.\d"	"19.9" et ".7" dans "a=19.9 b=.7"
+	L'élément précédent 1 ou plusieurs fois (greedy)	"be+"	"bee" dans "been" "be" dans "bent"
?	L'élément précédent 0 ou 1 fois	"rai?n"	"ran" et "rain" concordent
{n}	L'élément précédent exactement n fois	",\d{2}"	",12" en position 1 dans "7,1234"
{n,}	L'élément précédent au moins n fois (greedy)	"\d{2,}"	"12" et "456" dans "1 3 a12 b3 456"
{n,m}	L'élément précédent au moins n fois et au plus m fois (greedy)	"\d{3,5}"	"166" et "17668" concordent
*?	L'élément précédent 0 ou plusieurs fois (lazy : prend le minimum d'occurrences)		
+?	L'élément précédent 1 ou plusieurs fois (lazy)		
??	L'élément précédent 0 ou 1 fois (lazy)		
{n,}?	L'élément précédent au moins n fois (lazy)		
{n,m}?	L'élément précédent au moins n fois et au plus m fois (lazy)		

Greedy vs. Lazy : par défaut les quantificateurs sont "greedy", ils recherchent la sous-chaîne de taille maximale satisfaisant le motif. Si l'on ajoute un ? après le quantificateur, c'est la sous-chaîne de taille minimale satisfaisant le motif qui est cherchée (lazy).

Exemple :

Pattern = "<a>.*<a>" Chaîne = "x=<a>b<a>c<a>d<a>e"
La sous-chaîne qui concorde est "<a>b<a>c<a>d<a>"

Pattern = "<a>.*?<a>" Chaîne = "x=<a>b<a>c<a>d<a>e"
Deux sous-chaînes concordent "<a>b<a>" en position 2 et "<a>d<a>"

Note: quand une concordance a été trouvée, l'analyse reprend dans la sous-chaîne restante

Pattern = "<a>.*?<a>" Chaîne = "x=<a>b<a>c<a>d"

Une seule sous-chaîne concorde "<a>b<a>" en position 2 puisque dans la sous-chaîne restante "c<a>d" on ne retrouve plus le motif.

Jalons (Anchors) : commandes qui testent des repères particuliers correspondant à des concordances de taille nulle (ne correspond à aucune sous-chaîne) : début de mot, fin de mot, début de ligne, fin de ligne ...

^	La concordance doit avoir lieu en début de chaîne ou de ligne	"^d"	"1" dans "1a 2r"
\$	La concordance doit avoir lieu en fin de chaîne ou de ligne (avant \n)	"d{3}\$"	"-333" dans "-901-333"
\A	La concordance doit avoir lieu en début de chaîne		
\z	La concordance doit avoir lieu en fin de chaîne		
\G	La concordance doit avoir lieu là où la concordance précédente a eu lieu. Permet de vérifier que les concordances sont contiguës	"\G\d"	"(1)" "(3)" et "(5)" dans "(1)(3)(5)[7](9)"
\b	En dehors d'un groupe de caractères, teste la concordance avec le début ou la fin d'un mot (séparation entre un alphanumérique \w et un \W)	"\be"	"e" en positions 0 et 8 dans "ete tee etre"
\B	Négation de \b : n'est pas le début ou la fin d'un mot	"\Be"	"e" en positions 2,5 et 6 "ete tee"

Groupements

Dans les motifs reconnus, on peut isoler des groupes qui correspondent à des sous-chaînes de l'ensemble. Dans une chaîne concordant avec un pattern donné, il peut y avoir des sous-chaînes satisfaisant des parties du pattern complet. Les groupes sont délimités par des parenthèses

```
// Motif = 2 digits : 2 digits
// Les 2 paires de digits sont placées dans des groupes
Regex re = new Regex(@"(\d{2}):(\d{2})");
Match m = re.Match("13:24");
if (m.Success) {
    GroupCollection gc = m.Groups;
    for (int i = 0; i < gc.Count; i++) {
        Console.WriteLine(gc[i].Value);
        Console.WriteLine(gc[i].Index);
    }
}
```

Sortie

```
13:24
0
13
0
24
3
```

Dans cet exemple, on isole les deux paires de digits (avant et après le :). A chaque groupe dans le motif correspond une sous-chaîne dans la collection gc. Le groupe d'indice 0 est la chaîne concordant avec le pattern global. Le groupe d'indice 1 est le premier groupe (le premier jeu de parenthèses dans le motif). Le groupe d'indice 2 est le second groupe (le second jeu de parenthèses).

On peut aussi réutiliser les groupes dans les concordances du reste de la chaîne

```
/* \1 = 1er groupe Le motif est ici '2 paires identiques séparées par :' */
Regex re = new Regex(@"(\d{2}):(\d{2})");
MatchCollection mc = re.Matches("13:13 14:14");
//deux concordances "13:13" en position 0 et "14:14" en position 6
```


(subexpr)	Capture la sous-expression dans un groupe (repéré par un numéro : le premier groupe a le numéro 1, le suivant numéro 2 ...)	" (\d{2}) : (\d) "	"23:1" concorde "23" et "1" sont les groupes 1 et 2
\num	Fait référence à un groupe déjà reconnu et mémorisé précédemment (num entier)	" (\d) : \1" " (\d) : \d : \1" " (\d) (\d) \1\2"	"2:2" concorde "1:4:1" concorde "1212" concorde
(?<name>sexpr) \k<name>	Le groupe est nommé name. \k<name> représente le groupe nommé name	" (?<pa>\d{2}) : \d : \k<pa> "	"13:1:13" concorde
(?: subexpr)	Groupe sans capture (ne figure pas dans la collection Groups)		
(?= subex)	Requiert d'être suivi par le sous-motif subex sans le placer dans le match. Les concordances suivantes sont cherchées à la suite du match.	" \d (? = \d { 2 }) " Ne concordent que les digits suivis de 2 digits. L'analyse reprend à la fin du match précédent.	"1" "2" "4" et "5" dans "12344567"
(?! subex)	Négation : concorde seulement si le sous-motif subex ne suit pas	" \b (? ! un) \w + \b "	"sure", "used" dans "unsure sure unity used"
(?<= subex)	Requiert d'être précédé du sous-motif sans le placer dans le match.	" (? < = 19) \d { 2 } \b "	"99", "50", "05" dans "1851 1999 1950 1905 2003"
(?<! subex)	Négation :concorde seulement si le sous-motif subex ne précède pas	" (? < ! 19) \d { 2 } \b "	"51", "03" dans "1851 1999 1950 1905 2003"

Remarque (groupes nommés) : quand un groupe est nommé, on peut obtenir la sous-chaîne correspondant au groupe par son nom dans la collection **Groups**.

```
Regex re = new Regex(@"(?<h>\d{2}) : (?<m>\d{2})");
Match m = re.Match("12:24");
if (m.Success) {
    string s1 = m.Groups["h"].Value; // "12"
    string s2 = m.Groups["m"].Value; // "24"
}
```

Alternatives

	Concordance avec un des patterns séparés par	"th(e is at) "	"the" et "this" dans "this is the day. "
(?(exp) m_1 m_2)	Concordance sous condition. Si exp est trouvé, cherche si le motif m_1 est trouvé, sinon cherche si m_2 est trouvé. (exp est un sous-motif de m_1)	" (? (A) A \d \b \d { 3 }) " Si A est trouvé, cherche s'il est suivi d'un digit. Sinon, cherche s'il y a un mot commençant par 3 digits.	"A1", "910" dans "A10 C103 910"
(?(name) m_1 m_2)	(name) est un motif nommé (voir les groupements) Si le motif name est trouvé, on cherche si m_1 est reconnu. Sinon, on cherche si m_2 est trouvé		

1.9. Les flux IO de .NET (Stream)

Introduction – les flux de données

La notion de flux de données (stream) est une façon d'abstraire la lecture/écriture de données dans différents supports (mémoire, chaînes de caractère, fichiers, échanges de données en réseau ...)

Quel que soit le support des données, il existe un ensemble d'opérations de lecture/écriture qu'il est possible d'isoler et de décrire à travers une classe abstraite **Stream**. Ensuite, différentes classes concrètes fournissent le moyen effectif de lire/écrire sur des supports spécifiques. L'intérêt de l'abstraction **Stream** est de fournir des procédures communes d'exploitation des flux de données, en se souciant un peu moins des spécificités de chaque support.

Remarque (analogie avec STL/C++) : la bibliothèque STL fournit également une hiérarchie de classes de flux de données : **std::iostream** pour les entrées/sorties standard utilisateur, **std::fstream** pour les lectures/écritures dans des fichiers, **std::stringstream** pour la mémoire.

Tiré du site MSDN :

Les flux impliquent trois opérations fondamentales :

1. Vous pouvez lire à partir des flux. **La lecture** est le transfert de données d'un flux vers une structure de données tel qu'un tableau d'octets.
2. Vous pouvez écrire dans les flux. **L'écriture** est le transfert de données d'une structure de données vers un flux.
3. Les flux peuvent prendre en charge la **recherche**. La recherche consiste à envoyer une requête concernant la position actuelle dans un flux et à modifier cette dernière. La fonctionnalité de recherche dépend du type de magasin de stockage du flux. Par exemple, les flux de réseau n'ont pas de concept unifié pour une position actuelle et ne prennent donc généralement pas en charge la recherche.

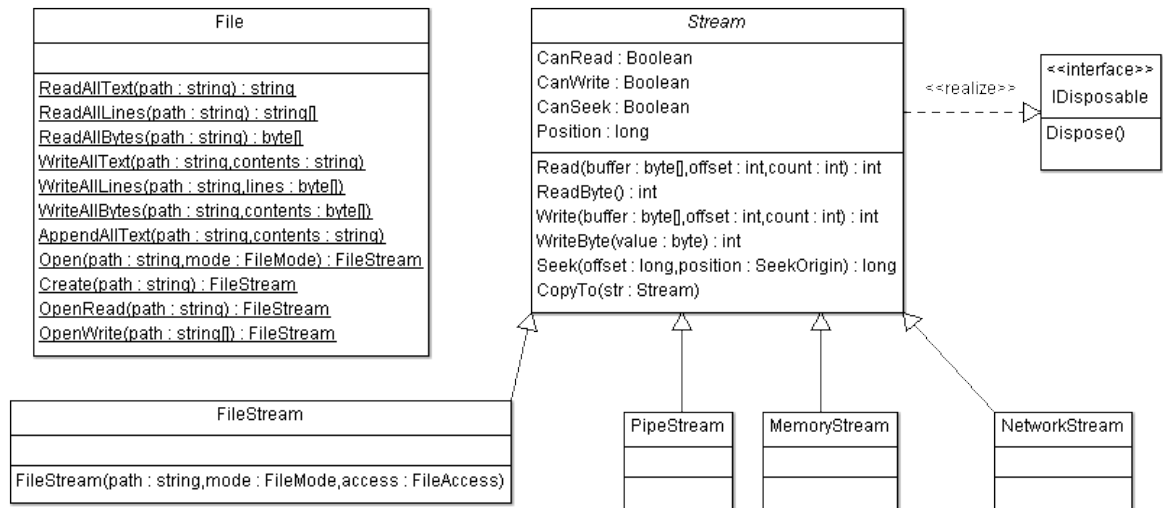
System.IO.Stream est la classe de base abstraite de tous les flux. Un flux est une abstraction d'une séquence d'octets, telle qu'un fichier, un périphérique d'entrée/sortie, un canal de communication à processus interne, ou un socket TCP/IP. La classe **System.IO.Stream** et ses classes dérivées donnent une vue générique de ces différents types d'entrées et de sorties, isolant le programmeur des détails spécifiques au système d'exploitation et aux périphériques sous-jacents.

Selon la source de données sous-jacente ou le référentiel, les flux peuvent prendre en charge certaines de ces fonctionnalités. Une application peut envoyer une requête à un flux concernant ses fonctionnalités en utilisant les propriétés [CanRead](#), [CanWrite](#) et [CanSeek](#).

Les méthodes [Read](#) et [Write](#) lisent et écrivent les données dans divers formats. Pour les flux qui prennent la recherche en charge, utilisez les méthodes [Seek](#) et [SetLength](#) ainsi que les propriétés [Position](#) et [Length](#) pour envoyer une requête concernant la position actuelle et la longueur d'un flux et les modifier.

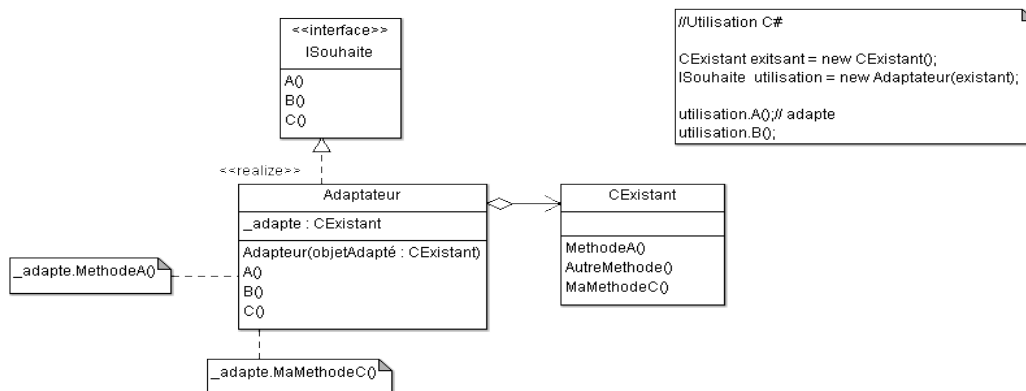
Hiérarchie/Structure des flux dans .NET

On présente dans un premier temps les différentes classes concrètes qui implémentent la classe **Stream**. La classe **FileStream** en particulier fournit l'accès aux fichiers. Notons tout de même que .NET fournit aussi une classe **File** constituée essentiellement de méthodes statiques permettant de lire/écrire dans des fichiers dont on fournit le chemin.



Adaptateurs de flux : les classes **Stream** gèrent la lecture/écriture essentiellement sous forme **byte** (octet). Pour gérer des types différents, il faut utiliser des adaptateurs de flux.

Le design pattern Adapter : quand on dispose d'une classe **CExistant**, et que ses méthodes ne nous conviennent pas totalement (on souhaite des noms de méthodes différents ou avec des paramètres différents), il est possible de décrire ce que l'on souhaite par l'interface **ISouhaite** et d'implémenter **ISouhaite** en redirigeant les appels vers un objet **CExistant**.



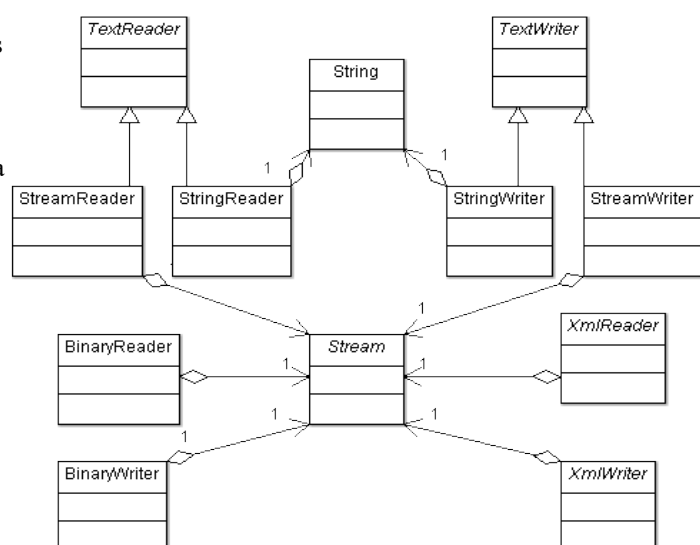
Ce principe d'adaptation est utilisé pour fournir différents moyens d'exploiter les flux. Les adaptateurs de flux sont décrits ci-contre sous forme de diagramme UML.

StreamReader : lecture d'un flux spécialisé pour la gestion du texte

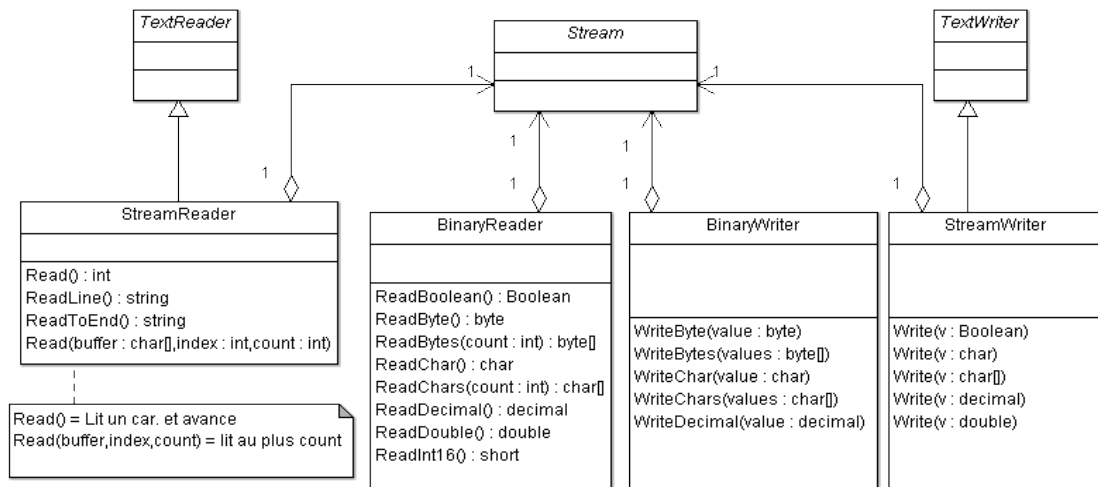
StreamWriter : écriture d'un flux spécialisé pour la gestion du texte

BinaryReader/BinaryWriter : permettent la lecture et l'écriture de différents types (int, float, bool ...) dans des flux.

XmlReader/XmlWriter : gestion des flux XML
Une description un peu plus détaillée des classes

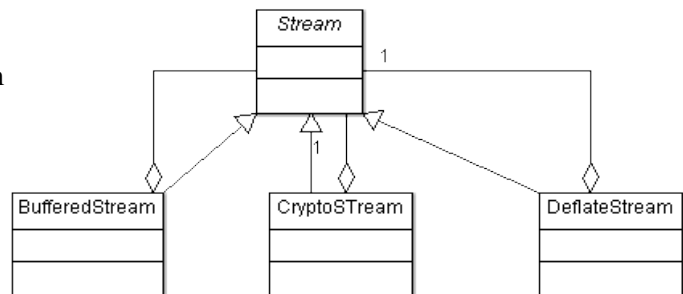


d'adaptation indique les méthodes supplémentaires pour gérer les types différents de byte.



Décorateurs de flux En plus des adaptateurs, .NET fournit des décorateurs de flux (au sens du design pattern Decorator). On peut donc décorer un flux de données (fichier, connexion réseau), par différents objets

CryptoStream : chiffrement des données
DeflateStream/GZipStream : compression/décompression des données
BufferedStream: ajoute couche de mise en mémoire tampon



Exemple (utilisation classe File) : affichage d'un fichier texte dans un contrôle TextBox

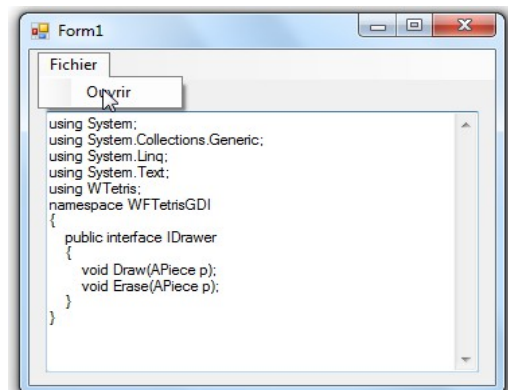
```

public partial class Form1 : Form
{
    public Form1 () {InitializeComponent();}

    private void ouvrirToolStripMenuItem_Click(object sender, EventArgs e)
    {
        OpenFileDialog dlg = new OpenFileDialog();
        if (dlg.ShowDialog() == DialogResult.OK)
        {
            try{
                textBox.Text = File.ReadAllText(dlg.FileName);
            }
            catch (Exception exc)
            {
                textBox.Text = exc.Message;
            }
        }
    }
}

```

Remarque (OpenFileDialog) : classe de boîte de dialogue pour chercher un fichier.



Exemple (utilisation classe FileStream)**Button A → Ecriture de N octets aléatoires dans data.bin****Button B → Lecture fichier binaire**

```

namespace WFFileStream
{
    public partial class Form1 : Form
    {

        private void buttonWrite_Click(object sender,
                                      EventArgs e)
        {
            Random r=new Random();
            // using (FileStream fstr = File.Create("data.bin")) [Alternative]
            using (FileStream fstr=new FileStream("data.bin",
                                                FileMode.Create,
                                                FileAccess.Write))

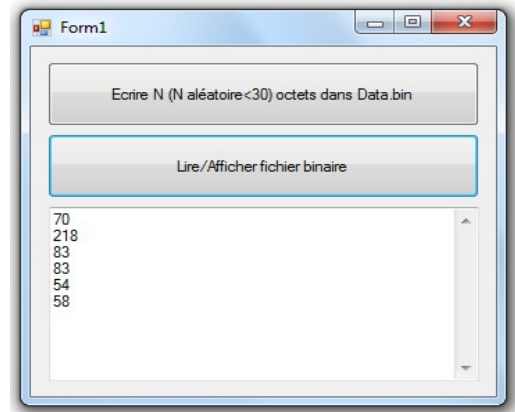
            {
                try{
                    byte[] data = new byte[r.Next()%20];
                    for (int i = 0; i < data.Length; i++) data[i] = (byte)r.Next();
                    fstr.Write(data, 0, data.Length);
                }
                catch (Exception exc){
                    textBox.Text = exc.Message;
                }
            }
        }

        private void buttonRead_Click(object sender, EventArgs e)
        {
            OpenFileDialog dlg = new OpenFileDialog();
            if (dlg.ShowDialog() == DialogResult.OK){
                try{
                    //using (FileStream fstr = File.OpenRead(dlg.FileName))
                    using (FileStream fstr = new FileStream(dlg.FileName,
                                                            FileMode.Open,
                                                            FileAccess.Read))

                    {
                        byte[] buffer = new byte[30];
                        int tailleLue = fstr.Read(buffer, 0, 30);
                        StringBuilder str=new StringBuilder();
                        for (int i = 0; i < tailleLue; i++){
                            str.Append(buffer[i].ToString() + "\r\n");
                        }
                        textBox.Text=str.ToString();
                    }

                }
                catch (Exception exc){
                    textBox.Text = exc.Message;
                }
            }
        }
    }
}

```



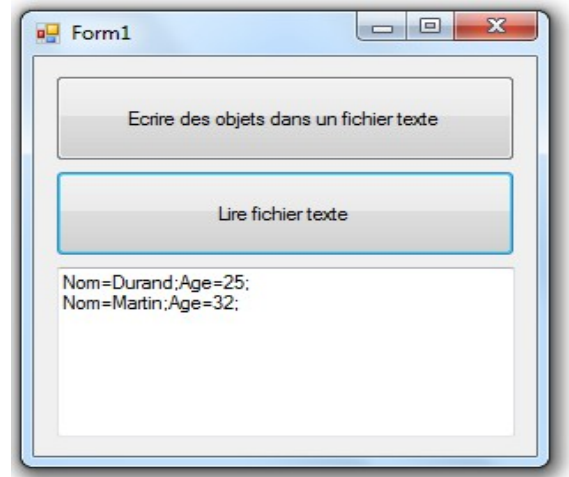
Exemple (utilisation classe StreamWriter/StreamWriter)

Les adaptateurs **StreamWriter** et **StreamReader** permettent de gérer les lectures écritures de type char/string dans des flux textes, donc aussi dans des fichiers. On décrit ici comment écrire, au format texte, la description d'objets **Personne (Nom/Age)** dans un fichier texte.

```
class Personne
{
    string nom;
    int age;

    public Personne(string N, int A) {
        nom = N; age = A;
    }

    public void WriteTo(TextWriter tw)
    {
        tw.Write("Nom="+nom+";");
        tw.Write("Age="+age.ToString()+";");
        tw.WriteLine();
    }
}
```



```
public partial class Form1 : Form
{
    ...
    private void buttonWrite_Click(object sender, EventArgs e) {
        SaveFileDialog dlg = new SaveFileDialog();
        if (dlg.ShowDialog() == DialogResult.OK)
        {
            try{
                using (StreamWriter str = new StreamWriter(File.Create(dlg.FileName)))
                {
                    Personne p1 = new Personne("Durand", 25);
                    Personne p2 = new Personne("Martin", 32);
                    p1.WriteTo(str);
                    p2.WriteTo(str);
                }
            }
            catch (Exception exc) {
                MessageBox.Show(exc.Message);
            }
        }
    }

    private void buttonRead_Click(object sender, EventArgs e)
    {
        OpenFileDialog dlg = new OpenFileDialog();
        dlg.Filter = "(*.txt)|*.txt";
        if (dlg.ShowDialog() == DialogResult.OK)
        {
            try{
                using (StreamReader str = new StreamReader(File.OpenRead(dlg.FileName)))
                {
                    textBox.Text = str.ReadToEnd();
                }
            }
            catch (Exception exc)
            {
                MessageBox.Show(exc.Message);
            }
        }
    }
}
```

Exemple (utilisation BinaryReader/BinaryWriter)

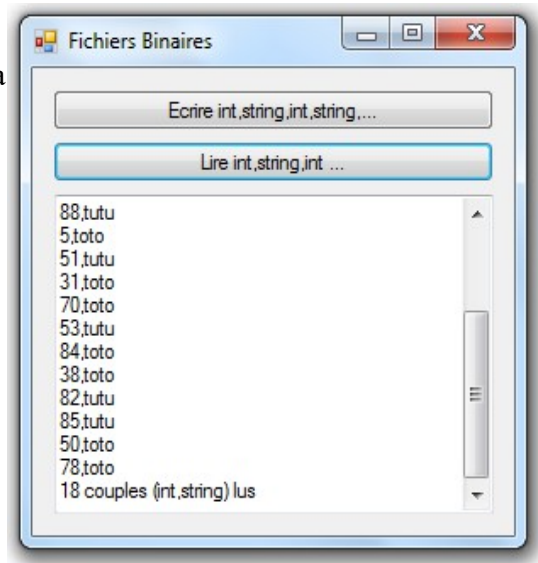
On écrit ici des suites de valeurs non 8bits. L'écriture et la lecture supposent que les données sont une alternance int, string, int, string ...

Les adaptateurs **BinaryReader/BinaryWriter** permettent la gestion binaire de données de taille supérieure à 1 octet. Pour un objet String, le stockage binaire est préfixé par la taille de la chaîne.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private void buttonEcrire_Click(object sender, EventArgs e)
    {
        Random r=new Random();
        int compte;
        string str;
        using(BinaryWriter bw=new BinaryWriter(File.Create("data.bin")))
        {
            compte=r.Next()%20;
            int i=compte;
            while(i>0)
            {
                bw.Write(r.Next()%100); // int32 compris entre 0 et 99
                if (r.Next() % 2 == 0) str="tutu";
                else str = "toto";
                bw.Write(str); // string "toto"/"tutu"
                i--;
            }
            textBox1.Text = compte.ToString() + " couples (int,string) écrits.";
        }

        private void buttonLire_Click(object sender, EventArgs e)
        {
            textBox1.Text = "";
            int compte = 0;
            int val;
            string str;
            using (BinaryReader br = new BinaryReader(File.OpenRead("data.bin")))
            {
                if (br.BaseStream.CanSeek) {
                    long taille = br.BaseStream.Length; // taille des données
                    while (br.BaseStream.Position != taille)
                    {
                        val = br.ReadInt32(); // lecture d'un Int32
                        str = br.ReadString(); // lecture d'un string
                        compte++;
                        textBox1.Text += val.ToString() + "," + str + "\r\n";
                    }
                    textBox1.Text += compte.ToString() + " couples (int,string) lus";
                }
            }
        }
    }
}
```



Description incomplète de la classe abstraite Stream

```

public abstract class Stream : MarshalByRefObject, IDisposable
{
    // true if the stream supports reading, seeking, time-out, writing
    public abstract bool CanRead { get; }
    public abstract bool CanSeek { get; }
    public virtual bool CanTimeout { get; }
    public abstract bool CanWrite { get; }

    // Retourne: A long value representing the length of the stream in bytes.
    // Exceptions : System.NotSupportedException (!CanSeek)
    public abstract long Length { get; }
    // Retourne : The current position within the stream.
    // Except. : IOException, NotSupportedException (if !CanSeek),
    //           ObjectDisposedException (if called after the stream was closed)
    public abstract long Position { get; set; }

    // Gets or sets a Read or Write time out value.
    public virtual int ReadTimeout { get; set; }
    public virtual int WriteTimeout { get; set; }

    // Closes the current stream and releases any resources (sockets, file ...)
    public virtual void Close();
    // Releases all resources used by the System.IO.Stream.
    public void Dispose();
    // Clears all buffers and causes any buffered data to be written
    // to the underlying device [Except.:IOException(if an I/O error occurs)].
    public abstract void Flush();

    // Reads a sequence of bytes from the current stream and advances
    // the position within the stream by the number of bytes read
    // buffer: when this method returns, the buffer contains the specified
    // byte array with the values between offset and (offset + count - 1) replaced
    // by the bytes read from the current source.
    // offset: The zero-based offset in buffer at which to begin storing the data read
    // count: The maximum number of bytes to be read from the current stream.
    // Retourne: The number of bytes read or zero (0) if the end has been reached.
    // Except.: ArgumentException(offset+count>buffer length),ArgumentNullException
    //           ArgumentOutOfRangeException(offset or count <0),IOException,
    //           NotSupportedException(!CanRead), ObjectDisposedException:
    public abstract int Read(byte[] buffer, int offset, int count);

    // Reads a byte from the stream and advances the position within the stream
    // by one byte, or returns -1 if at the end of the stream.
    // Except.: NotSupportedException(!CanRead),ObjectDisposedException:
    public virtual int ReadByte();

    // Writes a sequence of bytes and advances the current position
    // buffer,offset,count (see Read):
    // Except.: ArgumentException(offset+count>buffer length),ArgumentNullException
    //           ArgumentOutOfRangeException(offset or count <0),IOException,
    //           NotSupportedException(!CanWrite), ObjectDisposedException:
    public abstract void Write(byte[] buffer, int offset, int count);

    // Writes a byte to the current position in the stream and advances the position
    // Except.: IOException,NotSupportedException(!CanWrite),ObjectDisposedException
    public virtual void WriteByte(byte value);

    // Sets the position within the current stream.
    // offset: a byte offset relative to the origin parameter.
    // origin: of type System.IO.SeekOrigin, indicates the reference point
    // Retourne : the new position within the current stream.
    // Except.: IOException, NotSupportedException(!CanSeek), ObjectDisposedException
    public abstract long Seek(long offset, SeekOrigin origin);

    // Sets the length of the current stream.
    // value: The desired length of the current stream in bytes.
    // Except.: IOException, NotSupportedException(!CanWrite && !CanSeek)
    //           ObjectDisposedException:
    public abstract void SetLength(long value);

    public static readonly Stream Null; // A Stream with no backing store.
}

```


1.10. Communication réseau : Socket, NetworkStream, TcpClient, TcpListener

NA : pour toutes les notions du “réseau informatique” utilisées ici, ainsi que l'introduction aux classes .NET permettant d'exploiter les échanges réseau, la lecture du chapitre 9 du polycopié “Apprentissage du langage C# 2008 et du framework .NET 3,5” de Serge Tahé est vivement conseillée.

La communication entre applications à travers un réseau s'effectue le plus souvent avec l'interface standard **Socket** (appelé socket Berkeley). Un socket correspond à un point terminal d'un canal de communication entre deux applications. Chaque système d'exploitation (Linux, Windows etc.) fournit une implémentation de cette API pour faire communiquer des applications en s'appuyant sur différents protocoles. Le framework .NET permet d'exploiter l'API Socket à travers plusieurs classes.

Remarque : l'utilisation de l'API socket en langage C a été traitée en cours de “Réseau Informatique” en EI4 AGI. La présentation donnée ici s'intéresse essentiellement aux applications Client/Serveur suivant le protocole Tcp/IP.

System.Net.Sockets.Socket (namespace System.Net.Sockets/classe Socket)

La classe la plus proche de l'API Socket s'appelle **Socket** (namespace **System.Net.Sockets**). Cette classe fournit l'ensemble des méthodes permettant de créer et exploiter des sockets en choisissant le protocole de communication (notamment TCP et UDP). Cette classe Socket permet en particulier de créer un socket pour une application *Client* d'un Serveur Tcp/IP et de réaliser les appels aux fonctions sous-jacentes de l'API socket : connect, close, send, receive ...

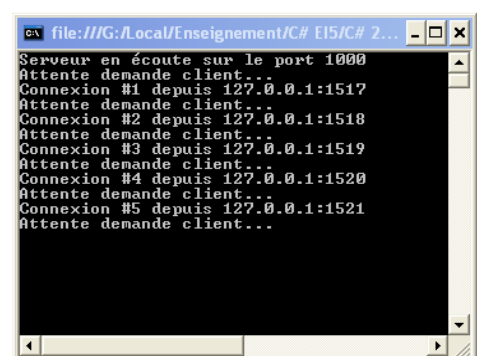
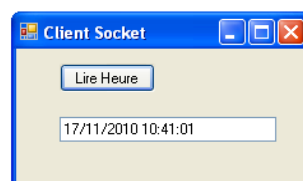
Exemple (Application Cliente utilisant un Socket et un flux NetworkStream)

Une *application Cliente* doit créer un socket et se connecter à un service TCP en fournissant l'adresse IP de la machine hébergeant le service ainsi que le numéro du port sur lequel le serveur attend les demandes de connexion.

Les flux, de manière générale, représentent la possibilité de lire/écrire/rechercher dans différents supports. Les flux permettant d'exploiter les échanges réseau sont décrits par la classe

NetworkStream. Un flux

NetworkStream s'appuie sur un socket sous-jacent pour les échanges de données à travers le réseau.



Pour l'exemple donné ici, on suppose qu'un serveur Tcp/IP est en attente de demandes de connexions (application console dont les sorties sont décrites ci-dessus). Pour chaque client, le serveur renvoie la date/heure courante sous forme d'une chaîne de caractères, puis ferme la connexion.

Nous décrivons ici l'écriture du client qui est une application WinForm où une connexion est établie sur chaque appui sur le bouton. Le contrôle textbox affiche la chaîne reçue par le client.

IPAddress : classe permettant de décrire une adresse IP et de parser une chaîne décrivant une adresse.

```
...
using System.Net.Sockets; // namespace pour classe Socket
using System.Net;         // namespace pour IPAddress ...
using System.IO;          // namespace pour les flux

namespace WFCClientSocketEx1
{
    public partial class Form1 : Form
    {
        public Form1() { InitializeComponent(); }

        /* méthode statique cherchant à réaliser une connexion au serveur
        d'adresse 'adr' et sur le numéro de port 'port'
        retourne null si échec de connexion */

        static private Socket ConnectSocket(IPAddress adr, int port)
        {
            IPEndPoint ipe = new IPEndPoint(adr, port);
            Socket tempSocket = new Socket(AddressFamily.InterNetwork,
                                           SocketType.Stream,
                                           ProtocolType.Tcp);

            tempSocket.Connect(ipe);
            if (tempSocket.Connected) return tempSocket;
            else return null;
        }

        private void buttonConnect_Click(object sender, EventArgs e)
        {
            IPAddress adr = IPAddress.Parse("127.0.0.1");
            try
            {
                // Socket implémente IDisposable
                using (Socket s = ConnectSocket(adr, 1000))
                {
                    if (s != null) // connexion OK
                    {
                        using (NetworkStream str = new NetworkStream(s))
                        {
                            {
                                using (StreamReader r = new StreamReader(str))
                                {
                                    char[] buffer = new char[100];
                                    r.Read(buffer, 0, 100);
                                    string msg = new string(buffer);
                                    textBox.Text = msg;
                                }
                            }
                        }
                    }
                }
            }
            catch (Exception exc)
            {
                textBox.Text = exc.Message;
            }
        }
    }
}
```

TcpClient/TcpListener (namespace System.Net.Sockets)

Le framework .NET fournit également des classes **TcpClient** et **TcpListener** pour simplifier l'écriture d'applications client/serveur en utilisant le protocole TCP. La création et le paramétrage des sockets sous-jacents sont cachés à l'utilisateur, ce qui simplifie un peu leur exploitation dans les cas courants. Un objet de la classe **TcpClient** se charge de créer un socket client pour communiquer avec un service TCP. Une fois le socket connecté, on peut obtenir un flux **NetworkStream** (**TcpClient.GetStream()**) pour échanger des données (en lecture/écriture) avec le serveur. On utilise ici l'adaptateur de flux **StreamReader** pour pouvoir extraire des chaînes de caractères du flux de données.

```
...
using System.Net.Sockets;
using System.IO;
using System.Net;

namespace WFTcPClient
{
    public partial class Form1 : Form
    {
        public Form1() { InitializeComponent(); }

        private void button1_Click(object sender, EventArgs e)
        {
            using (TcpClient client = new TcpClient())
            {
                try
                {
                    client.Connect(IPAddress.Parse("127.0.0.1"), 1000);
                    if (client.Connected)
                    {
                        using(NetworkStream nstr = client.GetStream())
                        using(StreamReader reader = new StreamReader(nstr))
                        {
                            char[] buffer = new char[100];
                            int tailleLue = reader.Read(buffer, 0, 100);
                            textBox.Text = new string(buffer, 0, tailleLue);
                        }
                    }
                }
                catch (Exception exc) {
                    textBox.Text = exc.Message;
                }
            }
        }
    }
}
```

Envoi/Réception de données entre Client et Serveur

L'objet **TcpClient** fournit un objet **NetworkStream** (flux) permettant d'échanger des données via le canal de communication établi entre client et serveur. L'écriture dans le flux revient à envoyer des données et la lecture du flux à lire des données reçues par le socket. L'exploitation d'un flux réseau reprend les principes des flux .NET en général (adaptateur et décorateur).

```

using (TcpClient client = new TcpClient("127.0.0.1", 9000))
{
    using (NetworkStream nstream = client.GetStream())
    {
        // envoi d'une trame avec 2 octets [0x00][0x03]
        Byte[] bufferEmission=new Byte[]{0x00,0x03};
        nstream.Write(bufferEmission, 0, bufferEmission.Length);

        // lecture des données reçues du serveur
        Byte[] bufferReception = new Byte[100];
        // retour : nombre d'octets effectivement lus
        int tailleLue = nstream.Read(bufferReception, 0, 100);
        // Remarque : si ici tailleLue == 100, il reste probablement
        // des octets dans le flux en lecture

        // EXPLOITATION DES DONNEES RECUES
        ...
    }
}

```

Elements pour l'écriture d'une application serveur Tcp/IP

Un objet de la classe **TcpListener** crée un socket d'écoute (qui attend les demandes de connexions client sur un port donné). Suite à une demande de connexion venant d'un client, il y a création d'un socket dit "de service" pour échanger des données avec le client. Il y a deux façons de traiter les clients.

Soit on traite les clients de façon séquentielle (le traitement d'un client ne peut pas débiter tant que le précédent n'est pas traité).

Soit, pour chaque demande, le serveur crée un socket de service et lance le traitement des échanges client/serveur dans un autre thread pour que le serveur redevienne disponible pour les prochains clients. Le traitement des clients est alors en parallèle.

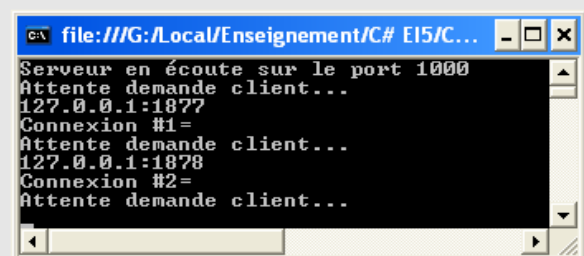
Serveur avec traitement séquentiel des clients (d'après ".NET2 et C#2", P. Smacchia, O'Reilly)

```

using System.Net;
using System.Net.Sockets;
using System.IO;

namespace ServeurConsole
{
    class Program{
        static void Main(string[] args)
        {
            TcpListener ecoute = null;
            try
            {
                ecoute = new TcpListener(IPAddress.Any, 1000);
                ecoute.Start();
                Console.WriteLine("Serveur en écoute sur le port 1000");
                TcpClient tcpClient = null;
                int nbDemandesServies = 0;
                while (!Console.KeyAvailable)
                {
                    Console.WriteLine("Attente demande client...");
                    tcpClient = ecoute.AcceptTcpClient(); // bloquant
                    string sRP =tcpClient.Client.RemoteEndPoint.ToString();
                    nbDemandesServies++;
                }
            }
            catch { }
        }
    }
}

```



```

        Console.WriteLine("Connexion #{0} depuis {1}", nbDemandesServices, sRP);

        Traite(tcpClient.GetStream());
    }
    catch (Exception ex) {
        Console.WriteLine("Exception :" + ex.Message);
    }
    finally {
        ecoute.Stop();
    }
}

static void Traite(NetworkStream ns) {
    try {
        using (StreamWriter writer = new StreamWriter(ns)) {
            writer.AutoFlush = true;
            writer.WriteLine(DateTime.Now.ToString());
        }
        ns.Close();
    }
    catch (Exception ex) { Console.WriteLine("Exception : " + ex.Message); }
}
}
}

```

Serveur avec traitement parallèle des clients ((d'après "Apprentissage du langage C# 2008 et du framework .NET 3,5" de Serge Tahé))

```

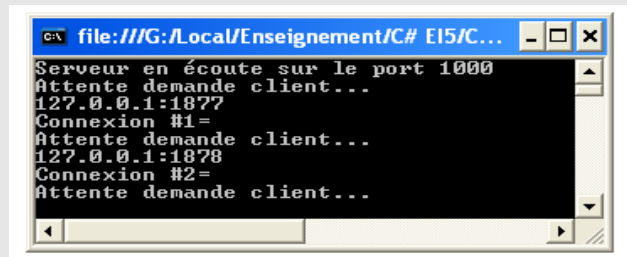
...
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.IO;

namespace ServeurConsole
{
    class Program
    {
        static void Main(string[] args)
        {
            TcpListener ecoute=null;
            try
            {
                ecoute = new TcpListener(IPAddress.Any, 1000);
                ecoute.Start();
                Console.WriteLine("Serveur en écoute sur le port 1000");
                TcpClient tcpClient = null;
                int nbDemandesServices = 0;

                while (!Console.KeyAvailable) {
                    Console.WriteLine("Attente demande client...");
                    tcpClient = ecoute.AcceptTcpClient(); // bloquant
                    Console.WriteLine(tcpClient.Client.RemoteEndPoint.ToString());

                    ThreadPool.QueueUserWorkItem(Service, tcpClient);
                    nbDemandesServices++;
                    Console.WriteLine("Connexion #{0}=", nbDemandesServices);
                }
            }
            catch (Exception ex) { Console.WriteLine("Exception :" + ex.Message); }
            finally {

```



```

        ecoute.Stop();
    }
}

public static void Service(Object infos) // service associé au Thread
{
    TcpClient client=infos as TcpClient;
    if(client!=null){
        try{
            using (TcpClient tcpClient = client)
            using (NetworkStream networkStream = tcpClient.GetStream())
            using (StreamReader reader = new StreamReader(networkStream))
            using (StreamWriter writer = new StreamWriter(networkStream)){
                writer.AutoFlush = true; //flux non bufferisé
                string reponse = DateTime.Now.ToString();
                writer.WriteLine(reponse);
            }
        }
        catch(Exception ex){Console.WriteLine("Exception:"+ex.Message);}
    }
}
}

```

1.11. La sérialisation binaire

La *sérialisation* est l'opération qui transforme la description de l'état d'un objet en une séquence de valeurs. La *désérialisation* est l'opération inverse qui permet à un objet d'obtenir son état à partir d'une séquence de données. Avec le framework .NET, il y a plusieurs moteurs de sérialisation. On peut réaliser notamment une sérialisation binaire (l'état de l'objet est décrit par une séquence d'octets) ou une sérialisation XML (l'état de l'objet est décrit au format XML).

A quoi sert la sérialisation ? Une fois sérialisé, l'état d'un objet peut être soit sauvegardé en vue d'être reconstitué plus tard (sérialisation dans un fichier), soit transmis via un flux de donnée pour que l'objet soit reconstitué dans une application distante (sérialisation dans un flux réseau).

Le mécanisme de sérialisation lorsqu'il est associé à une sauvegarde en fichier, permet également d'avoir des objets persistants : les objets peuvent être régénérés à partir d'un fichier et peuvent donc exister au delà de l'arrêt de l'application (les bases de données peuvent également servir à rendre les objets persistants!).

Sérialisation binaire

Le premier moteur de sérialisation, dans les cas généraux où l'on ne souhaite pas intervenir dans le processus de sérialisation, est très simple à utiliser.

En C#, il suffit de marquer la classe avec l'attribut **[Serializable]**.

```
[Serializable]
class Personne
{
    public string Nom;
    public int Age;
    public Personne(string nom, int age){    Nom = nom;    Age = age; }
    public override string ToString() {
        return Nom + ":" + Age.ToString() + " ans";
    }
}
```

Dès lors, il est possible de sérialiser/désérialiser un objet **Personne** à partir d'un flux de données. L'appel du mécanisme de sérialisation nécessite un objet *formatter*. Seuls deux formateurs sont disponibles :

BinaryFormatter et SoapFormatter

Les namespaces utilisés pour ces différentes classes sont

```
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization.Formatters.Soap;
using System.IO; // flux IO
```

Ex : Sérialisation vers un flux FileStream (avec BinaryFormatter):

```
Personne p1 = new Personne("Dupont", 24);
BinaryFormatter format = new BinaryFormatter();
using(FileStream fs = new FileStream("ObjetPers.dat",
                                    FileMode.OpenOrCreate,
                                    FileAccess.Write))
{
    format.Serialize(fs,p1); // sérialise l'objet p -> fichier ObjetPers.dat
}
```

Ex: Désérialisation depuis un flux FileStream (avec BinaryFormatter):

```

BinaryFormatter format = new BinaryFormatter();
using(FileStream fs = new FileStream("ObjetPers.dat",
                                   FileMode.Open,
                                   FileAccess.Read))
{
    // reconstitue l'objet p2 depuis ObjetPers.dat
    Personne p2 = format.Deserialize(fs) as Personne;
}

```

Les collections (**Array**, **ArrayList**, **List<>**,...) de types sérialisables sont sérialisables et les types primitifs .NET sont sérialisables

Ex : Sérialisation d'un tableau d'objets sérialisables

```

Personne[] tab= new Personne[4];
tab[0]=new Personne("Dupont", 24);
tab[1]=new Personne("Durant", 18);
tab[2]=new Personne("Martin", 13);
tab[3]=new Personne("Hardouin", 13);
BinaryFormatter format = new BinaryFormatter();
using(FileStream fs = new FileStream("ObjetPers.dat",
                                   FileMode.OpenOrCreate,
                                   FileAccess.Write))
{
    format.Serialize(fs,tab);
}

```

Ex: Désérialisation d'un tableau d'objets

```

BinaryFormatter format = new BinaryFormatter();
using(FileStream fs = new FileStream("ObjetPers.dat",
                                   FileMode.Open, FileAccess.Read))
{
    Personne[] tab = format.Deserialize(fs) as Personne[];
}

```

Remarque : dans une application Document/Vues ou Modele/Vues/Contrôleur, la partie Document ou Modèle (qui stocke les données métiers) peut être sérialisée dans un fichier.

Sérialisation XML (namespace `System.Xml.Serialization`)

NA : ce moteur de sérialisation est moins simple à mettre en oeuvre. En outre, une alternative est d'exploiter les données XML à travers des adaptateurs de flux XML.

Le moteur de sérialisation fourni via la classe `XmlSerializer` permet de décrire l'état d'un objet au format XML. Ce moteur ne nécessite pas de marquer les membres par des attributs, en revanche, seuls les champs publics sont sérialisés (ce qui limite un peu l'utilisation de ce moteur).

```

public class Personne
{
    public string Nom;
    public int Age;

    public Personne() { }
}

```


C#/.NET

```
public Personne(string N, int a){ Nom=N; Age = a; }

public override string ToString(){
    return "Nom=" + Nom + " \tAge=" + Age.ToString();
}
}
```

La sérialisation s'effectue à l'aide d'un objet **XmlSerializer**.

```
try
{
    XmlSerializer xSerial = new XmlSerializer(typeof(Personne));

    using (Stream fs = File.Open("personne.xml", FileMode.OpenOrCreate,
                                FileAccess.Write))
    {
        Personne p = new Personne("toto",12);
        xSerial.Serialize(fs,p);
    }
}
catch (Exception exc)
{
    // ...
}
```

Le résultat produit au format XML est

```
<?xml version="1.0"?>
<Personne xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Nom>toto</Nom>
  <Age>12</Age>
</Personne>
```

La désérialisation reconstitue un objet depuis un flux XML

```
XmlSerializer xSerial = new XmlSerializer(typeof(Personne));
using (Stream fs = File.OpenRead("personne.xml"))
{
    Personne p =(Personne)xSerial.Deserialize(fs);
}
```

1.12. Introduction à GDI+

GDI+ = Graphical Device Interface : La bibliothèque GDI+ est l'interface logicielle qui permet la gestion de sorties graphiques pour dessiner dans les fenêtres Windows. Ce document constitue une très brève introduction.

GDI+ permet

- le dessin de lignes, courbes, formes géométriques ... avec pinceaux et brosses
- la gestion des couleurs
- l'affichage d'images aux formats BMP, JPG, PNG, GIF, TIFF, WMF
- gestion de transformations de coordonnées à l'aide de matrices de transformations

Classe **System.Drawing.Graphics**

La classe **System.Drawing.Graphics** fournit le contexte, les outils de base et les méthodes pour dessiner

Pour dessiner, on doit obtenir un objet **Graphics** qui va servir d'interface de sortie graphique. Ensuite, cet objet contient un grand nombre de méthodes pour dessiner des lignes, des formes géométrique

Sont également couramment utilisées les classes ou structures ci-dessous:

Point : décrit un point à coordonnées entières

PointF : à coordonnées réelles

Size : dimension en X et Y (notamment pour les fenêtres)

Rectangle : défini par exemple par un **Point** et un objet **Size**

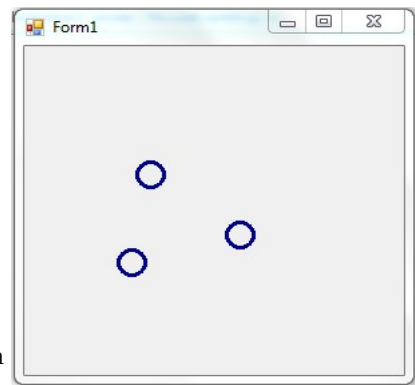
Pen : classe décrivant la couleur, le style, l'épaisseur d'un trait

Brush : classe décrivant la couleur et le motif de remplissage d'une zone

Color : décrit une couleur

Dans l'espace de nom **System.Drawing**, on peut obtenir des éléments graphiques pré-définis, notamment des couleurs (**Color.DarkBlue**), des pinceaux (**Pens.Black**) et des brosses (**Brushes.Chocolate**).

Premier exemple, à chaque événement **MouseClick** dans le formulaire, on dessine un cercle aux coordonnées du click.



```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    Graphics g = CreateGraphics();
    Pen p = new Pen(Color.DarkBlue, 3);
    g.DrawEllipse(p, new Rectangle(e.Location, new Size(20, 20)));
    g.Dispose();
    p.Dispose();
}
```

Interface IDisposable : dans .NET, une classe implémente l'interface **IDisposable** s'il est nécessaire que l'utilisateur restitue des ressources utilisées par l'objet. Cette interface ne contient qu'une seule méthode **Dispose** que l'utilisateur doit appeler (cf exemple ci-dessus). Différents objets graphiques implémentent cette interface, la classe **Graphics** en particulier.

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
    using(Graphics g = CreateGraphics())
    using (Pen p = new Pen(Color.DarkBlue, 3))
    {
        g.DrawEllipse(p, new Rectangle(e.Location, new Size(20, 20)));
    } // g.Dispose() et p.Dispose() appelés ici
}
```

Dessiner dans les fenêtres Lorsque l'on dessine à l'aide d'un objet **Graphics** fourni par une fenêtre, il n'est possible de dessiner qu'à l'intérieur de la zone décrite par cette fenêtre. Chaque fenêtre fournit un contexte de dessin qui lui est propre.

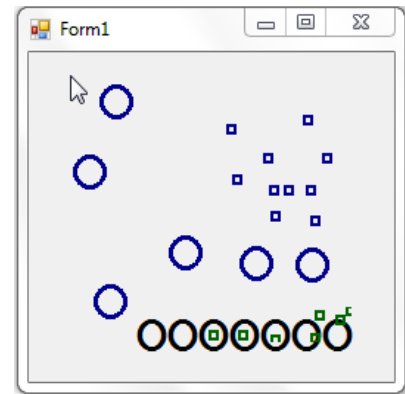
Illustration : dessiner dans la zone cliente du formulaire ou dans un contrôle

Placer un contrôle **panel** et un contrôle **label** dans le formulaire de l'application. Le formulaire et ses contrôles peuvent créer des contextes graphiques. Sur le click souris, on dessine différemment sur chaque contrôle.

```
private void Form1_MouseClick(object sender,
                             MouseEventArgs e)
{
    using (Graphics g = CreateGraphics())
    using (Pen p = new Pen(Color.DarkBlue, 3)) {
        g.DrawEllipse(p,
                     new Rectangle(e.Location,
                                   new Size(20, 20)));
    }
}

private void label_MouseClick(object sender, MouseEventArgs e)
{
    using (Graphics g = label.CreateGraphics())
    using (Pen p = new Pen(Color.DarkGreen, 2)) {
        g.DrawRectangle(p, new Rectangle(e.Location, new Size(5, 5)));
    }
}

private void panel_MouseClick(object sender, MouseEventArgs e)
{
    using (Graphics g = panel.CreateGraphics())
    using (Pen p = new Pen(Color.DarkBlue, 2)) {
        g.DrawRectangle(p, new Rectangle(e.Location, new Size(5, 5)));
    }
}
```



Le dessin avec GDI+ n'est pas persistant

Si l'on masque une partie de la zone graphique, le dessin s'efface. Lorsque l'on réalise des sorties graphiques, il faut prévoir de pouvoir ré-afficher le dessin, par exemple dans une surcharge de la méthode `OnPaint` d'un contrôle, ou dans la gestion de l'événement `Paint`

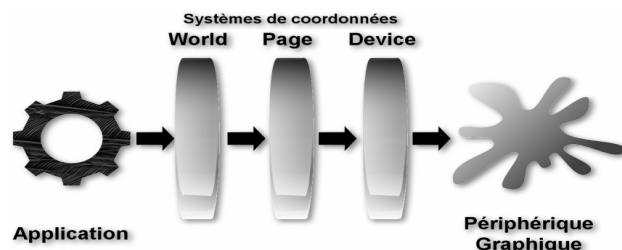
```
public partial class Form1 : Form
{
    public Form1() { InitializeComponent(); }

    protected override void OnPaint(PaintEventArgs e) {
        e.Graphics.FillRectangle(Brushes.Olive,
                                new Rectangle(new Point(10, 10),
                                              new Size(50, 60)));
        base.OnPaint(e);
    }
}
```

Système de coordonnées avec GDI+

Par défaut, les coordonnées que l'on fournit sont en pixels. Mais il existe en réalité trois systèmes de coordonnées qui se superposent.

Côté application, les coordonnées fournies sont dites en *World Coordinates*. Une première



transformation (*World Transformation*) les fait passer dans un second système de coordonnées (*Page*). Une seconde transformation (*Page Transformation*) transforme dans les coordonnées périphérique (device).

Matrice de transformation Chaque transformation décrite par une matrice de transformation est affine, c'est-à-dire une transformation linéaire suivie d'une translation. Une transformation affine est décrite par une matrice (objet **Matrix**) 3 lignes 2 colonnes.

Classe **System.Drawing.Drawing2D.Matrix**

Cette classe décrit une matrice de transformation affine (3x2). La création d'un objet **Matrix** par défaut est la

matrice identité c'est à dire

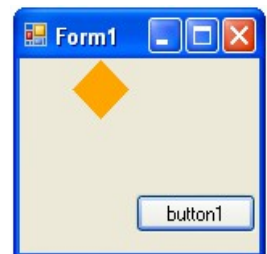
$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

Ensuite, les opérations de transformation se ramènent à des produits matriciels réalisés par les méthodes **Matrix.Rotate(angle)**, **Matrix.Translate(Dx,Dy)**, **Matrix.RotateAt(angle,PointF)**, **Matrix.Scale(sX,sY)**, **Matrix.Shear(sX,sY)**

Définition de la *World Transformation* par une matrice

```
using(Graphics g = CreateGraphics())
{
    Matrix m = new Matrix(); // M=Id
    m.Rotate(45); // M=Rot*M
    m.Translate(5,0,MatrixOrder.Append); // M=M*Tr
    g.Transform = m; // World Transform = M

    g.FillRectangle(Brushes.Orange,
        new Rectangle(new Point(0, 0),
            new Size(5,5)));
}
```



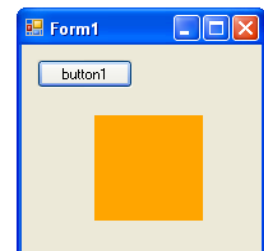
Transformations *Page Transformation*

Les transformations de type Page Transformation sont réalisées au moyen des propriétés

Matrix.PageUnit : de type GraphicsUnit
Matrix.PageScale : de type float

Exemple :

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.PageUnit = GraphicsUnit.Millimeter; // coordonnées en millimetres
    g.PageScale = 1.5F; // echelle 1,5
    g.FillRectangle(Brushes.Orange,
        new Rectangle(new Point(10,10), new Size(15,15)));
    base.OnPaint(e);
}
```



1.13. Table des matières

1. Compléments (Applications à fenêtres, Sérialisation, GDI+).....	2
1.1. Structure d'une application Windows.....	3
1.2. Accès aux contrôles dans le code.....	5
1.3. Gestionnaires d'événements.....	5
1.4. Même gestionnaire pour plusieurs événements.....	6
1.5. Formulaire (Fenêtre) Supplémentaire.....	7
1.6. Boîte de Dialogue (Formulaire Modal).....	8
1.7.  Application MDI	11
1.8. Les Expressions Régulières (classe Regex).....	13
1.9. Les flux IO de .NET (Stream).....	18
1.10. Communication réseau : Socket, NetworkStream, TcpClient, TcpListener.....	25
1.11. La sérialisation binaire.....	31
1.12. Introduction à GDI+	34