

Prise en main rapide de iTextSharp

par Joël Marty

Date de publication : 26 septembre 2008

Dernière mise à jour :

Article de présentation et de prise en main de la librairie de création de PDF: iTextSharp

1 - Introduction.....	3
1.1 - ITextSharp.....	3
1.2 - Concepts.....	3
2 - Operations sur les documents.....	4
2.1 - Format des pages.....	4
2.2 - Marges.....	4
2.3 - En-têtes / Pieds de page, métadonnées et initialisations.....	4
2.3.1 - A propos de l'objet « PdfWriter ».....	5
2.3.2 - En-têtes / Pieds-de-page.....	5
2.3.3 - Métadonnées.....	5
2.3.4 - ViewerPreferences.....	5
2.3.5 - Cryptage.....	5
2.4 - A propos des pages.....	6
3 - Opérations sur le texte.....	6
3.1 - Le Chunk.....	6
3.1.1 - Style.....	6
3.1.2 - Exposants et indices.....	6
3.1.3 - Couleur de fond.....	6
3.2 - L'objet Phrase.....	6
3.2.1 - Lettres grecques.....	7
3.2.2 - Plusieurs polices dans une même phrase.....	7
3.3 - L'objet Paragraph.....	7
3.4 - Listes.....	7
3.5 - Images.....	8
3.6 - Tableaux.....	8
4 - PdfContentByte et objets graphiques.....	9
4.1 - Graphiques simples.....	9
4.2 - Texte par le PdfContentByte.....	10
4.3 - Transparence et PdfGState.....	10
4.4 - Transformation et matrices.....	10
5 - Conclusion et crédits.....	11

1 - Introduction

1.1 - iTextSharp


TextSharp est une librairie open-source en C# adaptée depuis iText par Gerald Henson permettant la création de fichiers PDF. Elle dérive d'iText, son « grand frère » en JAVA, écrite à l'origine par Bruno Lowagie. iTextSharp est maintenant maintenue principalement par Paulo Soares, qui travaille également à améliorer iText. Les langages C# et JAVA étant similaires et iText bénéficiant d'une plus grande population d'utilisateurs, de l'aide peut-être trouvée facilement sur le site officiel d'iText et facilement adaptée à iTextSharp.

1.2 - Concepts

iText et iTextSharp introduisent certains concepts pour la manipulation de fichiers PDF. Le premier d'entre eux, et le plus important est le concept de « chunk ». On peut traduire ce mot par « unité textuelle ». Il s'agit en fait d'un objet comportant au choix du texte, des images, # ainsi que leurs propriétés (police, taille, transparence, orientation#). C'est l'objet principal d'iText et iTextSharp, celui dont dérivent tous les objets de plus haut niveau (phrases, paragraphes, #).

Un autre concept important porte sur le mode d'écriture des fichiers PDF : sous iTextSharp, vous devez en premier créer une instance de l'objet « Document » puis créer une ou plusieurs instances d'un « writer » qui interagit avec le document et écrit dans le flux de sortie (vous pouvez écrire dans un fichier ou dans un autre type de flux qu'un fichier pour utiliser iText dans le cadre d'une application client/serveur par exemple) et ouvrir le document nouvellement créé. A partir de là vous entrez dans la phase d'édition de document, dans laquelle vous ajoutez vos objets au document. Enfin, une fois vos objets ajoutés il faut fermer le document. Le flux de sortie est fermé et le fichier écrit sur le disque dur.

Vous avez un exemple de « Hello World » avec iTextSharp page suivante.

Ce document couvre l'utilisation d'iTextSharp au travers d'exemples. Il ne s'agit pas d'une documentation complète d'iTextSharp. Pour plus d'informations, consultez le site web d' [iTextSharp](#).

Exemple de programme console minimal avec iTextSharp.

```
using System;
using System.IO;
using iTextSharp;
using iTextSharp.text;
using iTextSharp.text.pdf;

namespace consoleapp
{
    class program
    {
        Static void Main(string[] args)
        {
            Document nouveauDocument = new Document();

            try
            {
                PdfWriter.GetInstance (nouveauDocument, new
                FileStream("#fichier.pdf#", FileMode.Create());
                nouveauDocument.Open();
                nouveauDocument.Add(new Phrase("#hello world#));
            }
            catch (DocumentException de)
            {
                Console.WriteLine("#error # + de.message);
            }
            catch (System.IO.IOException ioe)
            {
            }
        }
    }
}
```

```
        Console.WriteLine("#error # + ioe.message);  
    }  
    nouveauDocument.Close();  
}  
}
```

2 - Operations sur les documents

2.1 - Format des pages

Lorsque vous créez un document, vous avez accès à 3 constructeurs :

```
public Document ();  
public Document (Rectangle PageSize) ;  
public Document (Rectangle PageSize,  
    float marginLeft,  
    float marginRight,  
    float marginTop,  
    float marginBottom) ;
```

Le premier de ces trois constructeurs appelle en fait le second avec PageSize A4 en paramètre. Le second appelle le troisième avec une valeur de 36 en paramètre de chaque marge.

Ainsi en créant vos propres rectangles vous pouvez personnaliser l'apparence de vos pages.

Ex :

```
Rectangle pageSize = new Rectangle(144, 720) ;  
pageSize.BackgroundColor = new Color(0,255,255) ;  
Document document = new Document(pageSize);
```

Pour vous faciliter la vie, vous avez à disposition une collection de constantes pour définir la taille de la page. Pour une page au format paysage, utilisez la méthode rotate() de la propriété de la taille de la page :

```
Document document = new Document(PageSize.A4.rotate()) ;
```

2.2 - Marges

Les tailles de marges sont exprimées en points typographiques (ppp, points par pouces ou dpi, dots per inches). Le standard sous Windows est de 72ppp. Vous devrez donc faire un calcul pour créer des marges personnalisées.

Ex : la page A4 (21x29,7).

$21 / 2,54 = 8,2677$ pouces

$8,2677 \times 72 = 595$ points

$29,7 / 2,54 = 11,6929$ pouces ? 842 points.

La marge par défaut est de 36ppp (1/2 pouce) Pour une marge standard de 1,5 cm, cela correspond à 42 points.

NB : contrairement au changement de taille de page (qui prend effet sur la page suivante), le changement de marge est immédiat.

2.3 - En-têtes / Pieds de page, métadonnées et initialisations

La méthode open() de l'objet Document initialise certaines propriétés du document qui ne peuvent être changées par la suite. Ainsi, les métadonnées du document ainsi que les en-têtes et pieds de page sont à déclarer avant d'ouvrir le flux.

2.3.1 - A propos de l'objet « PdfWriter »

Lorsque vous utilisez cet objet, il peut être intéressant d'en utiliser plusieurs instances, pour écrire par exemple un morceau de texte avec une certaine police et un autre dans une autre police. Pour faciliter cette approche, vous avez à disposition deux méthodes de l'objet PdfWriter : pause() et resume(). Lorsque vous utilisez pause(), le PdfWriter sur lequel il s'applique ignorera alors tous les ordres du type document.add() laissant un autre PdfWriter s'en charger. En utilisant resume(), le PdfWriter recommencera à « écouter » les ordres document.add().

2.3.2 - En-têtes / Pieds-de-page

Il n'y a pas dans iTextSharp de méthode définie pour les en-têtes et pieds-de-page. Vous devez les construire vous-même en utilisant un tableau

2.3.3 - Métadonnées

L'ajout de métadonnées au document se fait au travers des méthodes définies ici :

```
public boolean AddAuthor(string author) ;
public boolean AddCreationDate();
public boolean AddCreator(string creator);
public boolean AddHeader(string name, string content);
public boolean AddKeywords(string keywords);
public boolean AddProducer();
public boolean AddSubject(string subject);
public boolean AddTitle(string title);
```

2.3.4 - ViewerPreferences

Vous pouvez définir des paramètres définissant le comportement du lecteur PDF à l'ouverture du fichier. Ceci se fait avec le writer (objet PdfWriter) :

```
PdfWriter writer = PdfWriter.GetInstance(document, new
FileStream(< fichier .pdf>));
writer.ViewerPreferences = PdfWriter.param
```

Avec pour param :

- Un layout : PageLayoutSinglePage, PageLayoutOneColumn#
- Un mode d'ouverture : PageModeUseThumbs, PageModeFullScreen#
- Des paramètres divers : HideToolBar, HideWindowUI, #

2.3.5 - Cryptage

Vous pouvez définir un cryptage pour le document permettant de limiter les actions de l'utilisateur vis-à-vis du document :

```
Writer.SetEncryption(force de cryptage, mdp utilisateur, mdp propriétaire,
autorisations)
```

Avec pour force de cryptage : PdfWriter.STANDARD_ENCRYPTION_xx avec xx = 40 ou 128 (cryptage sur 40 ou 128bits)

Et pour autorisations : PdfWriter.Allowxxxx

Avec pour xxxx : Printing, ModifyContent, Copy, ModifyAnnotations, FillIn, ScreenReader, Assembly, DegradedPrinting

Définir un mot de passe n'est pas forcément intéressant mais les autorisations peuvent l'être. Définissez mdp utilisateur et/ou propriétaire à null pour ne pas utiliser de mdp tout en conservant les autorisations.

2.4 - A propos des pages

vec iTextSharp, les sauts de pages sont gérés automatiquement, du moins lorsque du texte déborde d'une page. Si vous souhaitez créer une page arbitrairement, faites appel à la méthode `document.NewPage()` ;

NB : Si vous invoquez deux fois de suite cette méthode, aucune page vide ne sera créée. Vous pouvez ajouter un chunk vide pour simuler du contenu dans une page

3 - Opérations sur le texte

3.1 - Le Chunk

Comme expliqué précédemment, le chunk est l'objet primaire d'iText. Il peut contenir plus ou moins ce que vous voulez. Pour le texte, c'est un objet contraignant pour des opérations usuelles mais très intéressant pour des objets particuliers lorsque combinés avec d'autres objets : watermarks, images avec ou sans transformations, morceaux de texte particuliers#

Voici un exemple d'instanciation de chunk :

```
Chunk monChunk = new Chunk("#? Hello World #?", FontFactory.GetFont(FontFactory.COURRIER, 20,
Font.ITALIC, new Color(255, 0, 0)));
```

Cet exemple produit une unité contenant la chaîne « Hello World » écrit en rouge, dans la police COURRIER de taille 20 et en italique. Pour l'ajouter au document en cours il suffit de faire : `document.Add(monChunk)` ;

3.1.1 - Style

Vous voudriez peut-être écrire du texte dans plusieurs styles différents, en italique et gras par exemple ? Utilisez le caractère « | » pour pouvoir définir plusieurs propriétés à l'argument « style » de la méthode `GetFont()` de l'objet `FontFactory` (si vous utilisez `FontFactory`).

3.1.2 - Exposants et indices

L'objet `Chunk` possède la méthode `SetTextRise(float offset)` ; qui permet d'appliquer un offset au texte. Celui-ci peut-être positif ou négatif. Cependant, cette méthode ne fait qu'appliquer un offset et ne change pas la taille du texte. C'est à vous de le faire en instanciant un chunk d'une taille plus petite, lui appliquer un offset avec `SetTextRise()` et de l'ajouter à la suite de votre texte (A ce propos, vous avez ici un bon exemple de l'utilisation de plusieurs `writers` : un pour le texte normal et un pour les exposants).

3.1.3 - Couleur de fond

Vous pouvez changer la couleur de fond d'un chunk (effet de surlignage) avec la méthode `SetBackground(Color couleur)` ;

3.2 - L'objet Phrase

Vous pouvez voir l'objet `Phrase` comme une suite de chunks avec une information supplémentaire : l'espace entre les lignes (propriété `Leading`).

Une phrase possède une police principale mais vous pouvez définir arbitrairement une police différente pour des unités qui la composent :

```
Phrase maPhrase = new Phrase.GetInstance(#{? morceau de phrase 1??} ;  
maPhrase .Add(new Chunk(#{?morceau de phrase 2 #?} ) ;
```

Vous bénéficiez avec cet objet de méthodes d'insertions de caractères et chaînes, de fonctions de recherche, d'ajout, de copie#

3.2.1 - Lettres grecques

Vous pouvez écrire des lettres grecques dans vos phrases avec la méthode `GetInstance()` de la classe `Phrase`. Pour cela vous devez caster explicitement un nombre compris entre 913 et 970 (sauf 930) représentant l'alphabet grec dans la table Unicode.

```
Phrase grec = new Phrase.GetInstance(#{?j'ai un coefficient #? + (char)945} ;
```

3.2.2 - Plusieurs polices dans une même phrase

Si vous souhaitez composer des phrases contenant plusieurs polices, la méthode à employer est d'ajouter des `Chunks` de texte de polices différentes :

```
maPhrase.Add(new Chunk(#{?texte en times roman??}, new Font(Font.TIMES_ROMAN))) ;  
maPhrase .Add(new Chunk(#{?texte en courier #?},new Font(Font.COURIER))) ;
```

3.3 - L'objet Paragraph

L'objet `Paragraph` est une suite d'objets `Phrase` (ou `chunk`) avec en plus de la propriété `Leading`, une propriété `Indentation(Left/Right)` qui définit une indentation à gauche ou à droite du paragraphe ainsi qu'une propriété `Alignment` pour l'alignement.

```
Paragraph monParaph = new Paragraphe(objet) ;  
monParaph.Add(objet) ;  
monParaph.Leading = xxf ; (où xx est une valeur et f définit la valeur comme étant un float)  
monParaph.IndentationLeft = xxf ;  
monParaph.Alignment =Element.ALIGN_JUSTIFIED ;
```

Cet objet bénéficie également de propriétés telles `SpacingAfter` ou `SpacingBefore` qui définissent un espace vide avant ou après le paragraphe. Il hérite également des méthodes de l'objet `Phrase`.

3.4 - Listes

Afin de créer des listes numérotées ou à puces, vous disposez de l'objet `List`. Vous pouvez définir le comportement de la liste grâce aux multiples constructeurs qu'elle possède ou en modifiant ses propriétés.

Exemple:

```
List listenumerotee = new List(true, 20f) ;
```

Ou

```
List listeapuces = new List();  
List listeapuces.ListSymbol = #\u2022#; (code Unicode du symbole à utiliser comme puce)  
listeapuces.Add(#premiere ligne#);  
listeapuces.Add(#deuxieme ligne#);  
document.Add(listeapuces) ;
```

3.5 - Images

L'objet Image vous permet d'ajouter des images provenant de différentes sources : url, fichiers, données brutes, etc. Si vous souhaitez placer les images en des emplacements précis de la page, il vous faudra passer par l'objet PdfContentByte détaillé plus loin dans ce document. Voici comment s'utilise l'objet Image :

```
Image image1 = Image.GetInstance(#?image source??) ;  
Document.Add(image1) ;
```

3.6 - Tableaux

La création de tableaux avec iText est assez aisée grâce aux objets PdfPTable et PdfPCell. Leur utilisation est très aisée : construisez une table avec un certain nombre de colonnes et ajoutez-y des cellules.

```
PdfPTable tableau = new PdfPTable(3) ;  
PdfPCell cellule = new PdfPCell(new Paragraph(#?en-tête avec colspan de 3??)) ;  
cellule.Colspan = 3 ;  
tableau.AddCell(cellule) ;  
tableau.AddCell(#1.1#);  
tableau.AddCell(#1.2#);  
tableau.AddCell(#1.3#);
```

Dans cet exemple, si vous ajoutez plus de 3 colonnes elles seront automatiquement placées sur la deuxième ligne. Si vous placez moins de 3 cellules sur une ligne, celle-ci ne sera pas affichée (sauf si vous y insérez une cellule avec un colspan différent). Retenez donc que si vous construisez un tableau à n colonnes, chaque ligne devra contenir n cellules.

Si vous utilisez la méthode Document.Add() ; pour ajouter votre tableau au document, celui-ci sera affiché avec une taille égale à 80% de la taille de la page et sera affichée centrée. Pour définir votre propre largeur de tableau, vous devez définir 2 propriétés du tableau :

```
tableau.TotalWidth = xxx ; où xxx est la largeur du tableau (en points typo)  
tableau.LockedWidth = true ; sert à empêcher la méthode document.add de redimensionner le  
tableau.
```

Pour redimensionner chaque colonne indépendamment les unes des autres, vous devez remplir un tableau de float unidimensionnel de n cases où n est le nombre de colonnes de votre tableau. Chaque case du tableau représentant un pourcentage de la largeur du tableau. Vous lierez ensuite le tableau de float à votre objet PdfPTable au moment de sa construction ou à l'aide de la méthode SetWidths(float*+ ou int*+) ; :

```
float[] largeurs = {20,20,5,55} ;  
PdfPTable tableau2 = new PdfPTable(largeurs) ;  
table.AddCell(#?cellule1 : 20%??) ;  
table.AddCell(#?cellule2 : 20%??) ;  
table.AddCell(#?cellule3 : 5%??) ;  
table.AddCell(#?cellule4 : 55%??) ;
```

On redéfinit de nouvelles largeurs :

```
largeurs[0] = 0.1f ;  
largeurs[1] = 0.1f ;  
largeurs[2] = 0.4f ;  
largeurs[3] = 0.4f ;  
table.SetWidths(largeurs) ;
```

Pour créer des cellules de taille absolue, la méthode la plus facile (et la seule traitée dans ce document) est de combiner le tableau de largeur avec les propriétés TotalWidth et LockedWidth : si vous gardez le tableau largeurs défini ci-dessus et que vous définissez la largeur de votre tableau à 100, vous aurez deux colonnes de 10pts et 2 de 40.

NB : par défaut, les tableaux sont collés les uns aux autres sur un document. Utilisez les propriétés SpacingBefore et SpacingAfter pour espacer vos tableaux.

NB2 : Remarquez que dans la méthode AddCell(), une surcharge prend en paramètre une PdfPTable. Grâce à cette surcharge, vous pouvez créer des tableaux imbriqués :

```
Tableau2.AddCell(table1)
```

4 - PdfContentByte et objets graphiques

Pour l'instant nous n'avons utilisé que les fonctions de base d'iText sans nous préoccuper de la mise en page. En effet, lorsque nous ajoutons du texte, images ou objets aux pages, nous avons laissé iTextSharp s'occuper seul de la création des pages, mettre en forme le texte, etc., bien que nous ayons la possibilité de contrôler ces opérations.

Pourtant il est des cas où nous aimerions ajouter du contenu en des endroits bien précis, avec telle orientation, telle transparence s'il s'agit d'une image, telle couleur s'il s'agit de formes géométriques#

Le PdfContentByte s'utilise de la manière suivante :

```
PdfContentByte cb = writer.DirectContent ;  
cb.SetLineWidth(3f) ;  
cb.MoveTo(100, 700) ;  
cb.LineTo(200, 800) ;
```

Notez que le writer doit être créé avant d'initialiser le PdfContentByte. Par la suite vous n'avez qu'à utiliser les méthodes et propriétés du PdfContentByte pour ajouter des données à vos fichiers Pdf.

4.1 - Graphiques simples

Le PdfContentByte contient toute une collection pour dessiner des formes géométriques primitives :

```
cb.Circle(200f, 200f, 50f) ; arguments : position du centre en x et y et rayon  
cb.Rectangle(200f,200f, 50f, 100f) ; position x et y, largeur et longueur  
cb.SetColorFill(Color) ; définit la couleur qui sera utilisée pour remplir les formes.  
cb.Fill() ; si vous voulez que vos formes soient remplies.  
cb.SetColorStroke(Color) ; définir la couleur pour le contour des formes.  
cb.Stroke() ; dessine le contour des formes sans les remplir (contraire de Fill())
```

Les couleurs sont simples à gérer. Outre les couleurs intégrées dans le Framework iTextSharp, vous pouvez créer vos propres sets de couleurs avec le constructeur. L'utilisation est explicite et ne sera pas traitée ici.

Vous pouvez également dessiner des lignes. Pour cela, vous avez besoin de deux voire trois méthodes : cb.MoveTo(x,y) qui définit la position courante, cb.LineTo(x,y) qui dessine une ligne de la position courante définie par

MoveTo(x,y) vers la position passée en paramètre et cb.ClosePath() qui dessinera une ligne de la dernière position vers la position définie par le MoveTo(). Cela revient à fermer le chemin formé par les lignes.

4.2 - Texte par le PdfContentByte

L'utilisation du PdfContentByte pour la rédaction de texte est pratique lorsque vous voulez orienter du texte, selon un certain angle, provoquer un effet miroir ou d'autres transformations impossibles à faire avec les classes Chunk, Phrase ou Paragraph. Concrètement, vous devez dire à l'objet PdfContentByte lorsque vous passez en mode texte et lorsque vous en sortez. De plus, la gestion des polices est différente : vous devez instancier un objet BaseFont qui prend pour paramètres de constructeur un fichier de police sur le disque dur, une méthode d'encodage et une valeur booléenne indiquant si la police doit être incluse dans le fichier PDF de sortie (nécessaire dans le cas où il s'agit d'une police « exotique »). La procédure minimale à suivre est celle-ci :

```
cb.BeginText();
BaseFont bf = BaseFont.CreateFont("C:\\WINDOWS\\FONTS\\ARIAL.TTF", BaseFont.CP1252, true);
//on instancie une police Arial avec BaseFont
    cb.SetFontAndSize(bf, 10f); //on lui donne une taille de 10
cb.SetTextMatrix(200, 200); //on applique une matrice simple définissant les coordonnées du texte
    cb.ShowText("hello world"); //on demande l'affichage de du texte
cb.EndText();
```

On aurait pu demander une matrice plus complexe pour SetTextMatrix comprenant des rotations, etc. : cb.SetTextMatrix(0, 1, -1, 0, 200, 200); Cette méthode écrira un texte avec une rotation de 90° dans le sens trigonométrique Voir 4.4 pour plus d'infos sur les matrices de transformations.

4.3 - Transparence et PdfGState

Vous avez la possibilité de définir une transparence pour les objets créés avec le PdfContentByte. Pour cela, il vous faut créer un PdfGState qui est un dictionnaire des propriétés graphiques utilisées par le PdfContentByte. Bien que certaines propriétés soient directement accessibles par le PdfContentByte, d'autres sont réservées au PdfGState. Utilisation

```
PdfGState gs = new PdfGState();
gs.FillOpacity = 0.5f;
cb.SetGState(gs);
cb.Circle(300, 300, 75);
cb.Fill();
```

Ce code dessinera un cercle plein et transparent à 50%. Le PdfGState garde son état tant que vous ne lui avez pas dit de changer attention donc à l'utilisation. Pour plus de praticité vous avez la possibilité d'utiliser les méthodes cb.SaveState() et cb.RestoreState() pour sauvegarder un état donné du PdfGState sous forme de pile : vous devez avoir autant de SaveState() que de RestoreState() et le premier RestoreState appelle l'état enregistré par le dernier SaveState(). Si vous appelez RestoreState() avant SaveState(), cela lèvera une exception.

4.4 - Transformation et matrices

Vous pouvez appliquer des transformations géométriques aux objets générés par le PdfContentByte grâce à une matrice définie comme suit :

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

La troisième colonne est laissée en l'état car ici on ne s'intéresse qu'aux transformations 2D. Notez que les lettres a, b, c, d, e et f reviennent souvent dans iTextSharp et concernent les transformations géométriques de la même

manière qu'expliqué ici : Les paramètres e et f concernent les translations sur x-pixels et y-pixels respectivement. Si vous voulez agrandir ou réduire l'objet, vous devez agir sur les paramètres a et d. Le cas de la rotation est particulier étant donné que les rotations s'effectuent depuis l'origine (0,0), c.-à-d. depuis le coin bas gauche, vous devrez peut-être avoir à appliquer en plus une translation dans le cas où vous feriez « sortir » l'objet de la page. Exemple de rotation de 90° :

$$\begin{bmatrix} \text{Math.cos}(0) & \text{Math.sin}(1) & 0 \\ -\text{Math.sin}(1) & \text{Math.cos}(0) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notez que les valeurs d'angles sont en radians. Vous pouvez évidemment combiner toutes ces transformations en une seule matrice :

Où sX et sY représentent l'agrandissement et tX et tY la translation. Dans la pratique, il existe deux méthodes pour appliquer une transformation. La plus pratique est `cb.Transform(System.Drawing.Drawing2D.Matrix)` qui prend une matrice 2d en paramètre.

L'autre méthode est d'utiliser `cb.ConcatCTM(a, b, c, d, e, f)` qui prends donc les valeurs de la matrice (sous forme de float).



$$\begin{bmatrix} sX * \text{Math.cos}(angle) & sY * \text{Math.sin}(angle) & 0 \\ -sX * \text{Math.sin}(angle); & sY * \text{Math.cos}(angle) & 0 \\ tX & tY & 1 \end{bmatrix}$$

Astuce : Si le système d'axes ne vous plaît pas (origine en bas à gauche), vous pouvez changer l'origine en appliquant `cb.ConcatCTM(1f, 0f, 0f, -1f, 0f, PageSize.Ax.Height)` avec x la taille de la page (A4, A3, etc.) juste après avoir créé le PdfContentByte.

Nb : Notez que cette technique est assez difficile à utiliser dans certains cas. C'est pourquoi vous disposez de méthodes et propriétés concernant ces techniques directement dans la classe Image par exemple.

5 - Conclusion et crédits

Si vous voulez plus d'informations sur iTextSharp, je vous conseille de vous référer à la documentation iText (JAVA) car les noms de méthodes et propriétés sont les mêmes dans iText et iTextSharp et le fonctionnement est similaire.

Je vous conseille particulièrement  **iText by example** qui a également servi à construire ce document et vous donne l'adresse  **du tutorial iTextSharp** qui peut servir mais je vous déconseille de trop l'utiliser car il est basé sur une ancienne version d'iTextSharp et n'est plus maintenu.

Je remercie **Bruno Lowagie** et **Paulo Soares** qui m'ont aidé à mettre en place ce document.