

# TP C#10

## My ImageMagick

### Rendu

### Archive

Vous devez rendre un fichier zip avec l'architecture suivante:

```
rendu-tp-firstname.lastname.zip
|-- rendu-tp-firstname.lastname/
    |-- AUTHORS
    |-- README
    |-- MyImageMagick/
        |-- MyImageMagick.sln
        |-- MyImageMagick/
            |-- BitmapExtension.cs
            |-- ColorExtension.cs
            |-- MyImageMagick.cs
            |-- everything else except bin/ and obj/
```

- Vous devez bien sûr remplacer *firstname.lastname* avec votre propre login (qui est de la forme *prenom.nom*).
- Votre code doit compiler.
- Pour ce TP, vous êtes autorisés à implémenter d'autres fonctions que celles demandées, elles seront considérées comme des bonus. Cependant, vous devez garder une archive la plus légère possible.
- Ne rendez aucune image de test ! Nous testerons votre code avec nos propres images.

### AUTHORS

Ce fichier doit contenir une ligne formatée comme il suit : une étoile (\*), un espace, votre login et un retour à la ligne, RIEN DE PLUS.

Voici un exemple (où \$ représente un retour à la ligne et ↳ un espace):

```
*firstname.lastname$
```

Notez que le nom du fichier est AUTHORS avec AUCUNE extension.



Pour créer simplement un fichier AUTHORS valide, vous pouvez taper la commande suivante dans un terminal:

```
echo "* firstname.lastname" > AUTHORS
```

## README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un README vide sera considéré comme une archive invalide (malus).

## 1 Introduction

Nous vous demanderons pendant ce TP d'implémenter une petite bibliothèque de manipulation d'images (*ImageMagick*, *Photoshop*, etc.), ainsi qu'un programme pour l'utiliser en lignes de commande. Le temps que vous passerez sur ce TP dépendra considérablement de votre compréhension de la partie théorique et de votre rigueur. N'hésitez pas à prendre du temps sur cette partie avant de commencer à coder, soyez sûrs de comprendre les différents concepts, lisez la documentation si besoin, et posez des questions à vos ACDC.

### 1.1 Objectifs

Pendant ce TP, nous aborderons plusieurs notions, notamment :

- Manipulation d'images,
- Méthodes d'extension,
- Programmation fonctionnelle en C#,
- Interprétation basique d'arguments en ligne de commande.

## 2 Cours

### 2.1 Manipulation d'images

Vous connaissez sûrement déjà quelques notions basiques de manipulation d'images. Si ce n'est pas le cas, vous apprendrez durant cette semaine.



## Encodage des images

Pour commencer, vous devez savoir qu'une image est avant tout un fichier, rien de plus. Il s'agit d'une zone de votre mémoire, contenant des 0 et des 1. Dans la plupart des formats, le fichier commence par un en-tête contenant des informations générales sur l'image (taille, format, date de création, etc.). Après cet en-tête est l'encodage réel de l'image. L'encodage dépend du format choisi pour enregistrer le fichier, il en existe un grand nombre (PNG, JPG, BMP, SVG, PPM, etc.), le format est déterminé d'après le besoin de compresser l'image ou non, gérer la transparence ou non, et d'autres facteurs. Durant ce TP nous ne vous présenterons que le format BMP, qui est un des plus simples (mais qui est compatible avec d'autres comme le PNG ou le JPG via l'API C#).

### Format Bitmap (BMP) et modèle de coloration RVB

Nous allons traiter le format Bitmap 24 bits, qui ne gère pas la transparence. Dans ce format, une image n'est rien de plus qu'une matrice de pixels, la largeur et la hauteur de l'image sont donnés dans l'en-tête du fichier. Ce format utilise le modèle de coloration RVB pour encoder chaque pixel :

*Rouge, vert, bleu, abrégé en RVB ou en RGB, de l'anglais « Red, Green, Blue » est, des systèmes de codage informatique des couleurs, le plus proche du matériel. Les écrans d'ordinateurs reconstituent une couleur par synthèse additive à partir de trois couleurs primaires, un rouge, un vert et un bleu, formant sur l'écran une mosaïque trop petite pour être aperçue. Le codage RVB indique une valeur pour chacune de ces couleurs primaires.* – Wikipedia<sup>1</sup>

Dans le format BMP, chaque pixel est encodé sur 24 bits, 3 canaux de 8 bits (1 octet chacun). Chaque canal (octet) peut prendre une valeur dans l'intervalle [0, 255]. Chaque canal représente une certaine quantité de coloration (représenté par un int entre 0 et 255), rouge, vert ou bleue respectivement. Si les trois canaux ont une valeur de 0, le pixel apparaît noir, si les 3 valent 255, le pixel apparaît blanc, s'ils sont égaux, le pixel est gris, facile n'est-ce pas ?

### Le format Bitmap en C#

Dans la bibliothèque .NET, une classe a été implémentée pour simplifier la création, modification et lecture d'images Bitmap. Le nom de cette classe est étonnamment `Bitmap`<sup>2</sup>, n'hésitez pas à visiter la page MSDN de cette classe pour en apprendre plus à son propos. Cette classe vous permet de créer des objets Bitmap depuis des fichiers (JPG, PNG, BMP, et d'autres), de les éditer, de créer de nouvelles images à partir de ses dimensions, facilement accéder aux propriétés du fichier (largeur, hauteur, résolution, etc.), et enregistrer des objets Bitmap dans des fichiers. Bien entendu, elle vous permet aussi d'accéder à et de modifier des pixels en connaissant leur position dans la matrice. Les pixels sont stockés en tant qu'objets `Color`, cette structure sera expliquée après ce court exemple d'utilisation de la classe `Bitmap` :

<sup>1</sup>[https://fr.wikipedia.org/wiki/Rouge\\_vert\\_bleu](https://fr.wikipedia.org/wiki/Rouge_vert_bleu)

<sup>2</sup>[https://msdn.microsoft.com/en-us/library/system.drawing.bitmap\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.drawing.bitmap(v=vs.110).aspx)

```
1 // Créer une image depuis un fichier
2 Bitmap img = new Bitmap("path/to/file.png");
3
4 // Créer un magnifique fond d'écran HD vide
5 Bitmap empty = new Bitmap(1920, 1080);
6
7 // Sauvegarder l'image dans un autre fichier, dans un autre format
8 img.Save("new/path/to/file.jpg");
9
10 // Accéder facilement à la largeur et la hauteur
11 if (img.Width == img.Height)
12     Console.WriteLine("This picture is a square");
13
14 // Visiter tous les pixels, en remplaçant les blancs par des noirs
15 for (int i = 0; i < img.Width; ++i)
16 {
17     for (int j = 0; j < img.Height; ++j)
18     {
19         if (img.GetPixel(i, j) == Color.White)
20             img.SetPixel(i, j, Color.Black);
21     }
22 }
```

## Structure Color

La structure `Color`<sup>3</sup> vous permet de manipuler des couleurs et d'en créer des nouvelles depuis des valeurs RVB. Chaque objet `Color` possède 3 propriétés, R, G (pour « Green ») et B, et représentant les 3 canaux de couleurs du modèle de coloration RVB. Ces champs sont des entiers qui ne peuvent avoir une valeur que dans l'intervalle [0, 255]. Vous pouvez uniquement lire ces valeurs. Si vous souhaitez avoir une couleur avec des différentes valeurs, vous devez créer un nouvel objet, en utilisant la méthode statique `Color.FromArgb` (pas un constructeur, uniquement une méthode générant un `Color`). Cette structure possède aussi certains champs, représentants des couleurs spécifiques. Lisez l'exemple ci-dessous pour une petite utilisation de la structure, ou lisez la page MSDN correspondante :

```
1 // Créer une couleur depuis une existante
2 Color choco = Color.Chocolate;
3
4 // Récupérer les valeurs R, G et B et augmenter la luminosité
5 int r = choco.R + 50;
6 int g = choco.G + 50;
```

<sup>3</sup>[https://msdn.microsoft.com/en-us/library/system.drawing.color\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.drawing.color(v=vs.110).aspx)



```
7 int b = choco.B + 50;  
8  
9 // Vérifier que nous n'avons pas excédé la valeur limite de 255  
10 r = r > 255 ? 255 : r;  
11 g = g > 255 ? 255 : g;  
12 b = b > 255 ? 255 : b;  
13  
14 // Créer une nouvelle couleur depuis les nouvelles valeurs RVB  
15 Color brightChoco = Color.FromArgb(r, g, b);
```

Dans cet exemple, vous pouvez voir comment créer une **Color** (vous pouvez aussi en obtenir une d'un appel à `Bitmap.GetPixel`), augmenter la luminosité, et sauvegarder le résultat dans un nouvel objet couleur.

## Filtres

En graphisme numérique et manipulation d'image, un filtre est une altération (modification) effectuée sur une image. Les filtres les plus communs (vous les connaissez probablement) sont l'inversion (couleur opposée pour chaque pixel), flou, réglage de luminosité ou contraste, niveau de gris, etc. Vous allez devoir implémenter la plupart d'entre eux, et plus que vous ne connaissez probablement pas.

Par exemple, jetez un œil à Lena, et sa version inversée :



Figure 1: Lena et Lena inversée

## Lena

Mais qui est Lena ? Vous vous demandez sûrement. Lena, ou Lenna, est un modèle photographiée par Dwight Hooker pour une édition du magazine *Playboy* en novembre 1972. Cette photo d'elle est une *image de test pour les algorithmes de traitement d'image et est devenue de facto un standard industriel*

et scientifique<sup>4</sup>. Rien de plus spécial à propos de cette photo, c'est juste de la culture générale.

## 2.2 Dictionnaire

Les dictionnaires en C# sont des collections génériques implémentées dans l'espace de noms `System.Collections.Generic`. Ils permettent d'associer des clés d'un type à des valeurs d'un autre type. Quand vous ajoutez un élément à la collection, vous devez donc préciser une clé et une valeur. L'intérêt principal de cette structure de données est l'accès très rapide à une valeur en utilisant la clé associée : il s'agit d'un temps constant en utilisant une table de hachage. Vous pouvez aussi rechercher une valeur et récupérer sa clé, mais c'est plus lent (recherche linéaire). Un grand nombre de méthodes très pratiques utilisant les dictionnaires sont disponibles, nous vous encourageons à lire la page MSDN<sup>5</sup> pour en apprendre plus à ce sujet. L'exemple suivant associe des noms de jedi avec la couleur de leur sabre laser :

```
1 Dictionary<string, Color> jedi = new Dictionary<string, Color>;
2 jedi.Add("Obi-Wan", Color.Blue);
3 jedi.Add("Luke", Color.Green);
4 jedi.Add("Darth Vader", Color.Red);
5
6 Color c = jedi["Luke"];
7 // c == Color.Green. Cet utilisation est peu recommandée : si "Luke"
8 // n'était pas présent, une exception serait lancée.
9 // L'utilisation de TryGetValue est plus sûre
10 if (!jedi.TryGetValue("mevouc", out c))
11     Console.Error.WriteLine("mevouc n'est pas jedi référencé.");
12 else
13     // faire des trucs avec c
```

## 2.3 Méthodes d'extension

Les méthodes d'extension sont une façon commode d'ajouter des méthodes à un type, sans utiliser d'héritage ou modifier le type original. Elles sont définies comme des méthodes statiques, mais sont utilisées comme des méthodes membres d'un objet du type étendu. Une méthode est déclarée comme méthode d'extension en ajoutant en premier paramètre d'une méthode statique un objet du type étendu, précédé par le mot-clé `this`. Nous pouvons ensuite appeler notre méthode comme si celle-ci était interne au type, en utilisant la syntaxe `object.method()`. L'exemple suivant montre l'extension de la classe `Color` avec la méthode `IsBlack` :

```
1 public static void Main(string[] args)
2 {
```

<sup>4</sup><https://fr.wikipedia.org/wiki/Lenna>

<sup>5</sup>[https://msdn.microsoft.com/en-us/library/xfhwa508\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx)



```
3     Color c = Color.Black;
4     Console.WriteLine(c.IsBlack()); // Imprime True
5 }
6
7 public static bool IsBlack(this Color c)
8 {
9     return c.R == 0 && c.G == 0 && c.B == 0;
10}
```

Sachez qu'il ne s'agit que d'un sucre syntaxique : bien qu'elles soient utilisées comme des méthodes membres, elles restent des méthodes statiques prenant un objet en paramètre. Cela signifie que vous ne pouvez pas accéder aux attributs privées ou méthodes privées de la classe étendue.

## 2.4 Programmation fonctionnelle en C#

Au début de l'année, vous utilisez un langage fonctionnel, Caml, qui vous permet d'utiliser des fonctions comme d'autres variables, en les passant en arguments à d'autres fonctions, etc. Même si le C# est principalement un langage procédural et orienté objet, il propose depuis C#2.0 et C#3.0 des fonctionnalités fonctionnelles. La programmation fonctionnelle propose souvent des solutions plus élégantes pour des problèmes spécifiques.

### Les objets Func, Action, Predicate

Pour utiliser des fonctions comme des objets, C# propose 3 classes différentes : **Func**, **Action** et **Predicate**. Le plus simple pour comprendre leur utilisation sera avec un exemple :

```
1 Func<double, double> f = Math.Sin;
2 double y = f(4); // y = Math.Sin(4)
3 f = Math.Exp;
4 y = f(4); // y = Math.Exp(4);
```

Vous voyez ? Facile. Dans cet exemple nous définissons une fonction **f**, qui prend un double en argument, et retourne un double (le dernier type entre < et > est le type de retour). Nous assignons ensuite à **f** une référence à la méthode **Math.Sin**. Après cela, nous pouvons utiliser **f** comme nous utiliserions **Math.Sin**. Nous pouvons aussi changer pendant l'exécution la fonction représentée par **f**, c'est ce que nous faisons à la ligne 3 avec **Math.Exp**. Notez que nous ne pouvons uniquement assigner à une **Func** des méthodes ou fonctions qui ont le même nombre d'arguments, et le même type de retour et types d'arguments (ici **Math.Sin** et **Math.Exp** prennent toutes les deux un double et renvoient un double).

Une **Action** est simplement une **Func** renvoyant un **void**. Ainsi, en déclarant une **Action**, nous listons uniquement les types de ses arguments :



```
1 Action<string> println = Console.WriteLine;  
2 println("Test");
```

Un **Predicate** est un **Func** qui retourne un **bool**.

## Délégués

Parfois, vous ne voulez pas nécessairement utiliser des fonctions existantes, mais des sur-mesures, directement dans votre code. Vous pouvez faire cela en utilisant des délégués. Le mot-clé **delegate** à plusieurs sens, nous vous enseignerons ici comment l'utiliser pour créer des fonctions anonymes que vous pouvez assigner à des objets **Func** ou passer en argument à des méthodes. Jetez un œil à cet exemple :

```
1 Func<double, double> f = delegate(double x) { return 3 * x + 1; }  
2 double y = f(4); // y = 13
```

Comme vous voyez, cette déclaration est plus mathématique, et vous évite la définition d'une méthode pour une opération aussi simple (ici  $3 \times x + 1$ ). Nous n'avons pas besoin de donner un nom à la fonction que nous assignons ça **f**, ni donner le type de retour de cette fonction. Ce type de fonction anonyme est appelé délégué.

Le mot-clé **delegate** peut aussi être aussi utilisé pour définir vos propres types de délégués, et éviter l'usage de **Func** ou **Action**.

Si vous n'avez pas intégralement compris comment fonctionnent les délégués, nous vous recommandons GRANDEMENT de lire la page MSDN correspondante<sup>6</sup> ou demander à vos ACDC plus d'explications.

## Expressions lambda

Vous trouvez peut-être la syntaxe pour créer un délégué anonyme un peu trop verbeuse. Heureusement, les mainteneurs de C# ont créé une autre syntaxe, bien plus élégante. Le code suivant est absolument équivalent au précédent avec le mot-clé **delegate** :

```
1 Func<double, double> f = (x => 3 * x + 1);  
2 double y = f(4); // y = 13
```

Les parenthèses utilisées ici sont inutiles, elles servent uniquement à mettre en valeur la partie importante de cet exemple : l'expression lambda. Les expressions lambda sont uniquement un sucre syntaxique pour instancier des délégués d'une manière plus élégante. Chaque fois que vous utilisez une expression lambda, vous pouvez aussi utiliser la syntaxe avec le mot-clé **delegate**.

<sup>6</sup><https://msdn.microsoft.com/en-us/library/900ffyy8.aspx>

Jetez un œil à la page MSDN<sup>7</sup> pour avoir une présentation plus large de l'utilisation des expressions lambda.

### 3 Application : My ImageMagick

Durant ce TP, nous vous demanderons d'implémenter une version très simplifiée du logiciel de gestion d'image, *ImageMagick*<sup>8</sup>. Nous vous guiderons pour que vous puissiez développer un petit programme capable d'ouvrir une image, appliquer séquentiellement une suite de filtres sur l'image, avant de la sauvegarder dans un nouveau fichier. Voici un exemple de ce que vous serez capables de faire avec votre programme une fois terminé :

```
1 [sparrow@sea ~]$ ./MyImageMagick.exe img/lena.png --grayscale --invert  
2 --rotate-right --contrast 42 img/result.png
```



Figure 2: Lena et Lena modifiée

Le programme ouvre le fichier `lena.png`, applique dessus un niveau de gris, une inversion, une rotation, une augmentation du contraste de 42% (dans cet ordre) et sauvegarde l'image générée dans un fichier nommé `result.png`.

#### 3.1 Architecture

Cette application sera divisée en 3 classes (une par fichier) :

- `BitmapExtension`, cette classe statique contiendra tous les différents filtres et opérations applicables sur une image `Bitmap` donnée,
- `ColorExtension`, cette classe statique contiendra les opérations atomiques appliquées à des pixels,

<sup>7</sup><https://msdn.microsoft.com/en-us/library/bb397687.aspx>

<sup>8</sup><https://wwwimagemagick.org/>

- MyImageMagick, il s'agira de la classes contenant votre méthode Main, ainsi que toute la logique de votre application en ligne de commande.

La plupart (presque toutes) des méthodes que vous devrez implémenter seront des méthodes d'extension. Nous espérons que vous avez correctement compris la section correspondante dans la première partie du sujet, si ce n'est pas le cas, lisez-la de nouveau, consultez la documentation C# officielle<sup>9</sup>, ou demandez à vos ACDC.

### 3.2 Filtres basiques et opérations

Dans cette partie, vous allez implémenter des filtres généraux pour modifier des images Bitmap. Une ou deux seront un petit peu plus complexes, mais rien de vraiment difficile.

Tous ces filtres seront des méthodes d'extension de la classe `Bitmap` (Cette classe est `sealed`, nous ne pouvons donc pas en hériter). Les méthodes d'extension doivent être des méthodes statiques, implémentées dans une classe statique. La classe contenant les méthodes d'extension pour `Bitmap` sera `BitmapExtension` (nom original n'est-ce pas ?).

**Important:** Pour toute cette section ainsi que le reste du TP, toutes les méthodes d'extension de `Bitmap` devront être définies dans la classe `BitmapExtension`, et toutes celles de `Color` dans `ColorExtension`. Les méthodes mal placées ne seront pas évaluées !

#### Fais ceci sur tout cela

La plupart des filtres que vous pouvez implémenter sur des images ne sont qu'une répétition de la même modification appliquée sur tous les pixels de l'image. Pour ce faire, vous implémenterez une méthode `ForEachPixel`, cette méthode sera une extension de `Bitmap`, et prendra un délégué en argument. Le délégué sera un `Func`, prenant une `Color` en argument, et retournant la couleur modifiée. Voici le contenu de votre fichier `BitmapExtension.cs` après implémentation de cette méthode :

```
1 // Quelques usings
2 public static class BitmapExtension
3 {
4     public static void ForEachPixel(this Bitmap image,
5                                     Func<Color, Color> modify)
6     {
7         // Remplacer chaque pixel de l'image par
8         // le résultat de la modification
9     }
10    // Les autres méthodes d'extension de Bitmap seront implémentées ici
11 }
```

<sup>9</sup><https://msdn.microsoft.com/en-us/library/bb383977.aspx>

## Deviens un schtroumpf

Vous allez ici implémenter la méthode `Invert`, correspondant au filtre de la Figure 1. L'inversion d'une image est uniquement l'inversion de chacun de ses pixels, vous devrez donc utiliser votre `ForEachPixel`. Mais pour pouvoir l'utiliser, vous devez savoir comment inverser une `Color`.

Pour cela, vous allez commencer par implémenter la méthode d'extension `Invert` de la structure `Color`. Créez votre classe statique `ColorExtension`, et créez la méthode statique `Invert` :

```
1 // Quelques usings
2 public static class ColorExtension
3 {
4     public static void Invert(this Color color)
5     {
6         // FIXME
7     }
8     // Les autres méthodes d'extension de Color seront implémentées ici
9 }
```

Souvenez-vous, chaque canal de couleur a une valeur entre 0 et 255. Pour inverser une couleur, chaque canal doit être changé en son complémentaire dans cet intervalle : 0 devient 255, 255 devient 0, 128 devient 127.

Après l'implémentation de l'inversion de couleur, vous pouvez revenir à l'inversion de l'image, dans `BitmapExtension` :

```
1 public static void Invert(this Bitmap image);
```

Si vous avez correctement utilisé votre `ForEachPixel`, et l'inversion de couleur, cette méthode ne devrait pas prendre plus d'une seule ligne. Votre `ForEachPixel` doit prendre un délégué (`delegate` ou expression lambda), qui prend une `Color` et retourne son inverse.

## Retour aux années 50

Vous êtes un hipster. Les images colorées sont trop conventionnelles, le bon vieux temps vous manque, quand les images n'était juste qu'une question de luminosité. Pour cela, vous allez implémenter une méthode qui applique un filtre de niveau de gris sur une image. Vous allez suivre exactement le même procédé que pour le filtre précédent. Implémentez les 2 méthodes suivantes, dans les classes correspondantes :

```
1 public static void Grayscale(this Bitmap image);
2 public static void Grayscale(this Color color);
```

La première doit utiliser la deuxième ainsi que la méthode `ForEachPixel` et n'a aucune raison de faire plus d'une seule ligne.



Nous ne vous demandons pas d'implémenter un niveau de gris à base de simple moyenne, mais un niveau de gris qui prend en compte la perception des couleurs par l'œil humain (qui est plus aisément stimulé par le vert et le rouge que le bleu). Utilisez cette formule, correspondant au standard ITU-R BT.709<sup>10</sup> :

$$result = 0.21 \times red + 0.72 \times green + 0.07 \times blue$$

### Le côté lumineux vs. le côté obscur

À cause du maléfique Palplatine (aussi connu comme *Shepard*), vous êtes devenu de plus en plus sombre, mais parfois vous êtes capable de le combattre et de revenir vers la lumière. Pour ce faire, vous avez besoin d'ajuster votre luminosité.

Vous allez ici implémenter deux méthodes (comme vous l'avez fait plus tôt) pour modifier la luminosité d'une image en modifiant la luminosité de chacun des pixels :

```
1 public static void Brightness(this Bitmap image, int delta);  
2 public static void Brightness(this Color color, int delta);
```

`delta` est un entier représentant la quantité de lumière à ajouter (ou soustraire) à chaque couleur. Si cette valeur est supérieure à 255, l'image sera blanche, si elle est inférieure à -255, l'image apparaîtra noire.

Bien sûr, quand vous ajoutez la valeur `delta` à la valeur d'un canal, l'entier résultant ne doit pas être plus grand que 255, ni plus petit que 0. Nous vous recommandons fortement d'implémenter les méthodes suivantes dans votre classe `ColorExtension` :

```
1 private static int Restrict256(int n);  
2 private static int Restrict256(double n);
```

Elles vérifient uniquement si leur argument est entre les limites 0 et 255, si ce n'est pas le cas, la valeur limite la plus proche est renvoyée. Ces méthodes ne sont pas obligatoires, mais, mais seront très utiles pour éviter de réécrire les mêmes vérifications d'intervalle dans le reste du TP. La version prenant un `double` ne fait rien de plus que celle prenant un `int`, mais vous éviterez de devoir réfléchir à la conversion lorsque le résultat d'une opération vous donnera un `double`.

### Le noir est plus sombre, le blanc est plus clair

Désormais, vous ne voulez pas juste devenir sombre, ou lumineux. Vous désirez accentuer ces traits. Nous avons tous une partie de clair et de sombre en nous, nous voulons augmenter la différence entre ces deux côtés.

Implémentez les méthodes `Contrast` dans les classes correspondantes :

<sup>10</sup><https://fr.wikipedia.org/wiki/Luminance#Matri.C3.A7age>



```
1 public static void Contrast(this Bitmap image, int delta);  
2 public static void Contrast(this Color color, int delta);
```

La relation entre ces deux méthodes est la même que pour les questions précédentes, `delta` est supposé être entre -255 et 255 pour un résultat *normal*, mais des valeurs différentes doivent fonctionner, l'image résultante aura juste une drôle d'apparence.

Pour calculer le changement de contraste, la première étape est de calculer un facteur de correction, qui est donné par la formule suivante <sup>11</sup> :

$$F = \frac{259 \times (delta + 255)}{255 \times (259 - delta)}$$

Une fois ce facteur calculé, pour calculer la nouvelle valeur de chaque canal, utilisez la formule :

$$result = F \times (channel - 128) + 128$$

Soyez prudent, la valeur résultante peut être inférieure à 0 ou supérieure à 255, si vous l'avez implémentée, utilisez correctement votre méthode `Restrict256`.

### Transfert de dégradé

Vous allez ici implémenter un filtre un petit peu moins connu, le transfert de dégradé. Cette opération remplace chaque pixel par une valeur sur un dégradé, selon sa luminosité. Dans notre cas, nous représenterons le dégradé par 2 couleurs, le début et la fin de celui-ci. Par exemple, jetez un œil à cette image :

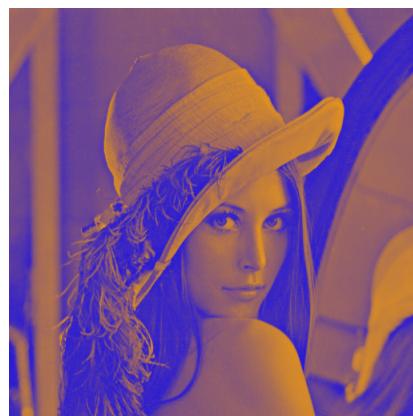


Figure 3: Transfert de dégradé bleu à orange sur Lena

<sup>11</sup><http://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-5-contrast-adjustment/>

Ceci est le résultat de l'application du transfert de dégradé entre bleu et orange sur la photographie de Lena. Tous les pixels sombres ont une valeur proche du bleu, les plus clairs une valeur proche du orange. Pour calculer les valeurs intermédiaires, vous devez créer une fonction affine (3 en réalité, une par canal), représentant l'évolution de la couleur résultante, lorsque la luminosité du pixel d'origine varie entre 0 et 255.

Considérez l'exemple théorique suivant pour une couleur à un seul canal (vous devrez faire cette opération 3 fois, pour chaque canal du modèle de coloration RVB) :

$$\Delta y = \frac{bright - dark}{255}$$

$$result = channel \times \Delta y + dark$$

*bright* est la couleur correspondant aux pixels blancs (orange dans l'exemple de Lena), *dark* est la couleur correspondant aux pixels noirs (bleu dans l'exemple).

Maintenant que vous avez compris comment ce filtre fonctionne, implémentez les deux méthodes, comme nous l'avons fait précédemment :

```
1 public static void GradientMap(this Bitmap img, Color dark, Color bright);  
2 public static void GradientMap(this Color col, Color dark, Color bright);
```

Comme d'habitude, la première appelle uniquement `ForEachPixel`, toute la logique de la transformation est implémentée dans la seconde méthode, modifiant des objets `Color`.

### Transforme moi !

Vous allez maintenant implémenter 4 méthodes d'extensions pour `Bitmap`, pour transformer l'image. Vous allez implémenter 2 rotations et 2 symétries.

```
1 public static void RotateRight(this Bitmap image);  
2 public static void RotateLeft(this Bitmap image);  
3 public static void SymmetryX(this Bitmap image);  
4 public static void SymmetryY(this Bitmap image);
```

La première et la seconde tournent l'image de 90°, dans le sens horaire et anti-horaire respectivement. Les dernières méthodes appliquent une symétrie sur l'image. `SymmetryX` applique une symétrie *horizontale* (haut-bas), et `SymmetryY` une symétrie *verticale* (droite-gauche). Soyez attentif de ne pas inverser les différentes méthodes (et attention à leur nommage).

**Indice :** Chacune de ces méthodes ne prend qu'une seule ligne. Lisez attentivement la page MSDN de la classe `Bitmap`<sup>12</sup> pour ne pas perdre de temps sur cet exercice.

<sup>12</sup>[https://msdn.microsoft.com/en-us/library/system.drawing.bitmap\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.drawing.bitmap(v=vs.110).aspx)



## Couvrez ce sein, que je ne saurais voir.

Dans cet exercice, vous allez apprendre à fusionner des images. La méthode que vous allez implémenter va prendre 2 images et une opacité, et générera la superposition des 2 images. Il n'y a rien de compliqué ici. La seule chose que vous devez savoir est que vous ne pouvez pas seulement ajouter les valeurs des pixels des 2 images, le résultat serait beaucoup trop clair, et ce n'est pas ce que nous voulons. Vous devez calculer la moyenne pondérée des 2 images, pour simuler une transparence réaliste.

Pour cela, vous allez implémenter 2 méthodes:

```
1 public static void Cover(this Bitmap a, Bitmap b, int opacity = 50);  
2 public static void Cover(this Color a, Color b, int opacity = 50);
```

L'argument `int opacity = 50` indique que le paramètre a une valeur par défaut. Si l'argument `opacity` n'est pas donné à la méthode, sa valeur vaudra 50. Ce paramètre est une valeur de pourcentage, dans chacune de ces 2 méthodes vous devez vérifier que le pourcentage est valide, et le corriger autrement (0 si la valeur était négative, 100 si elle était trop élevée).

La première méthode itère sur chaque pixel de chaque image, appelle la deuxième méthode sur chacun d'entre eux et les remplace. Si les 2 images ne font pas la même taille, vous devez lancer une `ArgumentException` avec un message approprié.

La méthode `Cover` pour les objets `Color` est une moyenne pondérée :

$$result = \frac{a \times (100 - opacity) + b \times opacity}{100}$$

Bien sûr, vous devez transposer la formule en C# pour chacun des 3 canaux de couleur.



Figure 4: `new Bitmap("lena.png").Cover(new Bitmap("fruits.png"), 60);`

Jetez un œil à cet exemple, nous recouvrions la photo de Lena, par une photographie de fruits<sup>13</sup>, avec une opacité de 60%. Notez que cela aurait été équivalent de couvrir l'image de fruits avec Lena et une opacité de 40%.

<sup>13</sup><http://www.birpo.com/image/categories/Fresh-Fruits.jpg>

N'essayez pas d'utiliser `ForEachPixel` dans cet exercice. La modification à faire ne dépendant pas uniquement de l'image source, cette méthode n'est pas applicable.

### 3.3 Génération de dégradé

Implémentez la méthode suivante :

```
1 public static void Gradient(this Bitmap image, Color left, Color right);
```

Cette méthode dessine un dégradé de la couleur *left* à la couleur *right* sur l'image donnée (masquant les pixels existants s'il y en a). Nous vous demandons un dégradé linéaire, de gauche à droite, ne réalisez rien de plus farfelu ici.

Voici une image générée avec cette méthode, comme exemple :

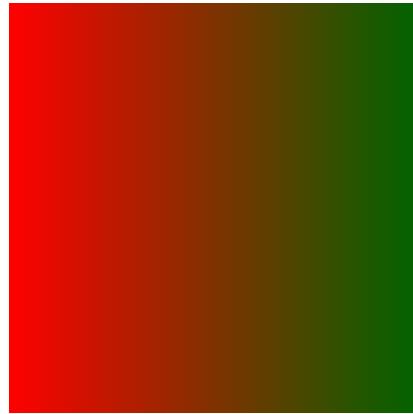


Figure 5: `new Bitmap(512, 512).Gradient(Color.Red, Color.DarkGreen);`

### 3.4 Extraction de palette

Nous allons maintenant vous demander d'implémenter la méthode d'extension suivante:

```
1 public static void Palette(this Bitmap image, int n)
```

Nous vous recommandons de lire l'exercice en entier avant de le commencer. Cette méthode va dessiner une palette des *n* couleurs les plus fréquentes de l'image d'entrée, en la découplant en *n* rectangles verticaux le rectangle le plus à gauche étant de la couleur la plus fréquente. Vous pouvez trouver ci-dessous un exemple de la palette des 5 couleurs les plus fréquentes dans l'image de cette chère Lena.

Chaque bande a pour largeur  $1/n^{me}$  de la largeur de l'image, et a pour longueur la hauteur de l'image. Mais bien sûr, il serait trop simple de prendre directement les couleurs de chaque pixel. En effet, dans une image, il y a souvent beaucoup de pixels ayant des couleurs très proches, mais avec d'infimes variations. Il serait intéressant de les prendre telles quelles pour notre palette : on pourrait se

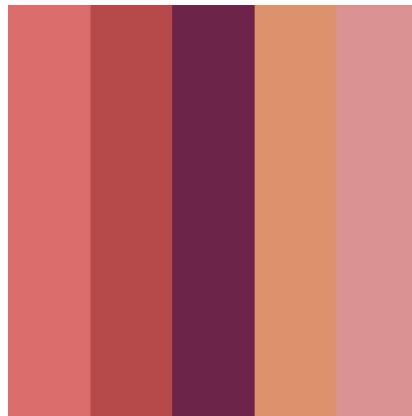


Figure 6: Palette des 5 couleurs les plus fréquentes dans la photo de Lena

retrouver avec une palette de couleurs quasiment unie avec des variations quasi imperceptibles. Nous allons donc utiliser un ensemble de couleur réduit sur lequel on va approximer les couleurs présentes dans notre image. Pour ce TP, nous avons choisi de nous limiter à un ensemble de 512 couleurs. Pour les calculer, on doit faire prendre à chacun de nos canaux une valeur parmi 8 possibles, entre 0 et 255 (sachant que 0 et 255 sont deux de ces valeurs). Utilisez des doubles lors de vos calculs pour avoir une précision maximale, et ne les convertissez en entier uniquement lors de la création de votre objet `Color`.

Une fois que vous avez calculé vos couleurs, vous allez avoir besoin d'approximer les couleurs de l'image à celles de votre ensemble. Pour faire cela, vous aurez besoin de calculer la différence entre deux couleurs. Voici une formule pour vous y aider :

$$\sqrt{(red1 - red2)^2 + (green1 - green2)^2 + (blue1 - blue2)^2}$$

Vous aurez en parallèle besoin de retenir le nombre d'apparitions de chacune des couleurs de l'ensemble. Pour faire cela, l'usage d'un dictionnaire `Dictionary<Color, int>` peut être une bonne idée.

Une fois tout cela fait, il ne vous reste plus qu'à prendre les n couleurs les plus fréquentes (Regardez MDSN si vous avez choisi d'utiliser un dictionnaire), et dessiner les rectangles.

### 3.5 Application en ligne de commande

Il est désormais temps de faire de ce TP un vrai programme. Nous allons vous demander de prendre des arguments depuis la ligne de commande (vous souvenez-vous du paramètre `string[] args` du `Main`?). Votre programme devra prendre au moins 3 arguments : le premier sera le chemin de l'image d'entrée et le dernier sera l'image de sortie. Entre les deux, vous aurez au moins un, mais un nombre potentiellement illimité d'options, correspondants à des traitements pour votre image. Les traitements doivent être exécutés séquentiellement. Par exemple la commande suivante :

```
1 [sparrow@sea ~]$ ./MyImageMagick.exe img/lena.png --grayscale --invert  
2 --rotate-right --contrast 42 img/result.png
```

Va charger l'image `img/lena.png`, la mettre en noir et blanc, inverser les couleurs de la Lena en noir et blanc, la faire pivoter vers la droite, et modifier le contraste de la Lena en noir et blanc inversée et pivotée, et sauvegarder le résultat dans `img/result.png`.

Voici la liste des options que votre programme doit reconnaître :

```
1 Usage: ./MyImageMagick.Exe source-image {options} destination-image  
2 Options  
3 -b, --brightness VALUE  
4 -c, --contrast VALUE  
5 -m, --cover PATH OPACITY  
6 -d, --gradient COLOR COLOR  
7 -a, --gradient-map COLOR COLOR  
8 -g, --grayscale  
9 -i, --invert  
10 -p, --palette NUMBER  
11 -l, --rotate-left  
12 -r, --rotate-right  
13 -x, --symmetry-x  
14 -y, --symmetry-y
```

Comme vous pouvez le voir, certaines options prennent des paramètres en plus (obligatoires). La plupart d'entre eux sont des nombres. Toutefois certaines prennent des couleurs. Pour celles-ci, nous nous attendons à ce que vous utilisez la méthode `Color.FromName` qui prend une string et renvoie un objet `Color` de type `KnownColor`<sup>14</sup>. Si les arguments donnés à votre programme sont invalides, vous devez afficher une message sur la sortie standard d'erreur et arrêter le programme. Vous ne devez pas cracher à cause d'une exception non gérée.

<sup>14</sup>[https://msdn.microsoft.com/en-us/library/system.drawing.knowncolor\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.drawing.knowncolor(v=vs.110).aspx)

## 4 Boni:

Plusieurs boni sont envisageables:

- Filtre de flou (gaussien, directionnel, etc.),
- Matrice de convolution,
- Filtre de seuil,
- Filtre de détection des bords,
- Ajout de texte style meme,
- Rognage,
- Redimensionnement,
- Filtres artistiques,
- Dégradé radial,
- Génération d'un bruit de Perlin,
- Afficher la palette comme un dégradé multicolore (arc-en-ciel),
- N'importe quoi d'étonnant.

Si vous en ajoutez, n'oubliez pas de les lister dans votre fichier **README**, ou ils ne seront pas évalués. De plus, ces boni sont des options additionnelles, et ne doivent pas modifier le comportement des filtres existants.

*The code is the law.*