TP C#10
My ImageMagick

Submission

Archive

You must submit a zip file with the following architecture :

```
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/
```

- You shall obviously replace *firstname.lastname* with your login (which is *firstname.lastname*).
- The code must compile.
- In this practical, you are allowed to implement any other methods than those specified, they will be considered as bonus. However you must keep the archive's size as light as possible.
- Do not submit any test image ! We will test your code with our own pictures.

AUTHORS

This file must contain the following : a star (*), a space, your login and a newline, NOTHING ELSE. Here is an example (where $ represents the newline and ␣ a blank space):

```
*␣firstname.lastname$
```

Please note that the filename is AUTHORS with NO extension.

To create your AUTHORS file simply, you can type the following command in a terminal:

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

README

You must write in this file any comments on the practical, your work, or more generally on your strengths / weaknesses, you must list and explain all the boni you implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

You will be asked during this practical to implement a small library of image processing (*ImageMagick*, *Photoshop*, etc.), and a program to use it in command line. The time you will spend on this practical will quite depend on your understanding of the theoretical part and your rigour. Do not hesitate to spend time on this part before starting to code, be sure to understand the different concepts, read documentation if needed, and ask questions to your ACDC.

1.1 Objectives

During this practical, we will approach new notions such as:

- Image processing,
- Extension methods,
- Functional programming in C#,
- Basic program arguments parsing.

2 Basic Knowledge

2.1 Image processing

You probably know a bit about image encoding already. If you do not yet, you will learn this week.

Image encoding

First, you have to know that an image is a file, nothing else. It is an area of your memory, containing 0 and 1. Usually, in most format, the file first starts with a header containing general information

about the picture (size, format, date of creation, etc.). After that, you have the actual image encoding. The encoding depends on the format chosen to save the file, many of them exist (PNG, JPG, BMP, SVG, PPM, etc.), the format is determined according to whether you want to compress your picture or not, handle transparency or not, and other factors. In this practical we will only present you the BMP format, which is one of the simplest (but is compatible with many others such as PNG or JPG through the C# API).

Bitmap (BMP) format and RGB color model

We will explain the 24 bits Bitmap format, which do not allow transparency. In this format, an image is just a matrix of pixels, the width and height of the picture are given in the header of the file. This format use the RGB color model to encode pixels:

The RGB color model is an additive color model in which red, green and blue light are added together in various ways to reproduce a broad array of colors. The name of the model comes from the initials of the three additive primary colors, red, green and blue. – Wikipedia[1]

In BMP format, each pixel is encoded on 24 bits, 3 channels of 8 bits (1 byte each). Each channel (byte) can have a value in the range $[0, 255]$. Each channel represents an amount of a color (represented by an `int` between 0 and 255), red, blue and green respectively. If all the three channels have a value of 0, then pixel is black, if they all are 255, the pixel is white, if they are the same, the pixel is gray, easy isn't it ?

Bitmap format in C#

In the .NET library, a class has been implemented to simplify creation, modification and reading of Bitmap pictures. The name of this class is surprisingly `Bitmap`[2], feel free to visit the corresponding MSDN page to know more about it. This class allows you to create Bitmap objects from existing files (JPG, PNG, BMP, and others) to edit them, create new images from nothing but the dimensions, easily get properties of the file (width, height, resolution, etc.), and save Bitmap objects to files. Of course, it also allows you to get and modify pixels by knowing their position in the matrix. The pixels are stored as `Color` objects, this structure will be explained just after this example of usage of the `Bitmap` class:

```
1 // Create image from a file
2 Bitmap img = new Bitmap("path/to/file.png");
3
4 // Create a beautiful empty wallpaper
5 Bitmap empty = new Bitmap(1920, 1080);
6
7 // Save the picture in another file, in a new format
```

[1] https://en.wikipedia.org/wiki/RGB_color_model
[2] https://msdn.microsoft.com/en-us/library/system.drawing.bitmap(v=vs.110).aspx

```
8  img.Save("new/path/to/file.jpg");
9
10 // Easily get width and height
11 if (img.Width == img.Height)
12   Console.WriteLine("This picture is a square");
13
14 // Visit all pixels of image, replacing white pixels by black pixels
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 }
```

Color structure

The `Color`[3] structure allows you to manipulate colors and create new ones from RGB values. Each `Color` object has 3 properties, R, G and B, representing the 3 color channels of the RGB color model. These fields are integers which can not have a value out of the range $[0, 255]$. You can only get these values. If you want to have a color with different values, you have to create a new object, using the `Color.FromArgb` static method (not a constructor, just a method which generates a `Color`). This structure also contains special fields, which represent particular colors. Read below a short usage example of this structure, or read the MSDN corresponding page:

```
1  // Create color from existing one
2  Color choco = Color.Chocolate;
3
4  // Get R, G and B values and increase the brightness
5  int r = choco.R + 50;
6  int g = choco.G + 50;
7  int b = choco.B + 50;
8
9  // Check that we did not exceed the limit value of 255
10 r = r > 255 ? 255 : r;
11 g = g > 255 ? 255 : g;
12 b = b > 255 ? 255 : b;
13
14 // Create a new color from the new RGB values
15 Color brightChoco = Color.FromArgb(r, g, b);
```

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

In this example, you can see how to create a color (you can also get it from a `Bitmap.GetPixel` call), increase the brightness, and save the result in a new color object.

Filters

In computer art and image processing, a filter is a modification done on image. The more common filters (you probably know them) are inversion (opposite color of each pixel), blur, contrast or brightness modification, gray scale, etc. You will be asked to implement most of them, and more that you might not know.

As an example, take a look to Lena, and the inverted version of herself:

Figure 1: Lena and inverted Lena

Lena

But who is Lena ? You may ask. Lena, or Lenna, is a model photographed by Dwight Hooker for an issue of the *Playboy* magazine in November 1972. This picture of her is *a standard test image of image processing since 1973*[4]. Nothing more special about it, just some geek culture information.

2.2 Dictionary

Dictionaries in C# are associative generic collections implemented in the `System.Collections.Generic` namespace. They allow you to associate keys of a type to values of another type. When you add an element to that collection, you must therefore precise a key and a value. The main interest of this data structure is the very fast access to a value using the key associated: it is constant in time through the use of a hash table. You may also search a value and find its key, but it's slower (linear search). A lot more of useful methods using dictionaries are available, we encourage you to read the MSDN page[5] to learn more about it. The following example associates jedi names with their lightsaber color :

[4] `https://en.wikipedia.org/wiki/Lenna`
[5] `https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx`

```
Dictionary<string, Color> jedi = new Dictonary<string, Color>;
jedi.Add("Obi-Wan", Color.Blue);
jedi.Add("Luke", Color.Green);
jedi.Add("Darth Vader", Color.Red);

Color c = jedi["Luke"];
// c == Color.Green. This usage is not recommended : if "Luke" were not
// present, an exception would've been raised.
// The usage of TryGetValue is safer
if (!jedi.TryGetValue("mevouc", out c))
  Console.Error.WriteLine("mevouc is not a registered jedi.");
else
  // do stuff with c
```

2.3 Extension methods

Extension methods are a convenient way to add methods to a type, without the use of inheritance or modifying the original type. They are defined as static methods, but are used as a member methods of an object of the extended type. They are created by putting in first parameter of a static method an object of the extended type, preceded by the keyword `this`. We can then call our method as if it was a member of the type, using the `object.method()` syntax. The following example shows the extension of the class `Color` with the method `IsBlack`:

```
public static void Main(string[] args)
{
  Color c = Color.Black;
  Console.WriteLine(c.IsBlack()); // Writes True
}

public static bool IsBlack(this Color c)
{
  return c.R == 0 && c.G == 0 && c.B == 0;
}
```

Be aware that it is only syntactic sugar: although they are used as member methods, they are still static methods taking an object in parameter. That means that you may not access private attributes or use private methods of the extended class.

2.4 Functional programming in C#

At the beginning of the year, you were coding using a functional language, Caml, which allows you to use functions like other variables, passing them as arguments to other functions, etc. Even if C# is mainly a procedural and object-oriented language, it provides since C#2.0 and C#3.0 some functional features. Functional programming often allows more elegant solution to specific problems.

Func, Action, Predicate objects

To use functions as objects, C# provides 3 different classes : `Func`, `Action` and `Predicate`. It will be simpler for you to understand how to use them with an example:

```
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);
```

You see ? Easy. In this example we define the function `f`, which takes a double as argument, and returns a double (the last type between `<` and `>` is the return type). We assign to `f` a reference to the method `Math.Sin`. After that, we can use `f` like we would be using `Math.Sin`. We can as well change during run time the function represented by `f`, this is what we do at line 3 with `Math.Exp`. Please note that we can only assign to a `Func` methods or functions with the same number of arguments, and the same return type and arguments type (here `Math.Sin` and `Math.Exp` both take a double and return a double).

An `Action` is simply a `Func` returning `void`. Therefore, when declaring an `Action`, we only list the types of its arguments:

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

A `Predicate` is a `Func` which returns a `bool`.

Delegates

Sometimes, you do not want to necessary use existing functions, but to use custom ones, directly in your code. You can do that using delegates. The `delegate` keyword has several meanings, we will teach you here how to use it to create anonymous functions that you can assign to `Func` objects or pass as argument to methods. Take a look to this example:

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

As you see, this declaration is more mathematical, and avoids you the definition of a method to do a simple operation (here $3 \times x + 1$). We do not need to give a name to the function we assign to `f`, neither give the return type of the function. This type of anonymous functions is called delegate.

The `delegate` keyword is also used to explicitly define your own delegates types, and avoid the use of `Func` or `Action`.

If you did not understand everything about delegates, we HIGHLY recommend you to read the corresponding MSDN page [6] or ask your ACDC to give you more explanations.

Lambda expressions

You might find the syntax to create an anonymous delegate a bit wordy. Fortunately, C# maintainers created an other syntax, much more elegant. The following example does exactly the same as the one before with the `delegate` keyword:

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

The parentheses used above are totally useless, they are only here to help you notice the point of this example: the lambda expression. Lambda expressions are just a syntactic sugar to instantiate delegates in a more elegant way. Every time you use the lambda expression syntax, you could also use the syntax with `delegate` keyword.

Take a look to the MSDN page [7] to have an overview of the usage of lambda expressions.

3 Application: My ImageMagick

In this practical, we will ask you to implement a very simplified and custom version of the image processing software, *ImageMagick*[8]. We will guide you so you can develop a small program that opens a picture, applies sequentially some filters on it, and saves it to a new file. Here is an example of what you will be able to do with your program once done:

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

The program opens the file `lena.png`, applies grayscale, inversion, rotation, contrast increasing of 42% (in that order) and saves the generated picture as a file named `result.png`.

[6] `https://msdn.microsoft.com/en-us/library/900fyy8e.aspx`
[7] `https://msdn.microsoft.com/en-us/library/bb397687.aspx`
[8] `https://www.imagemagick.org/`

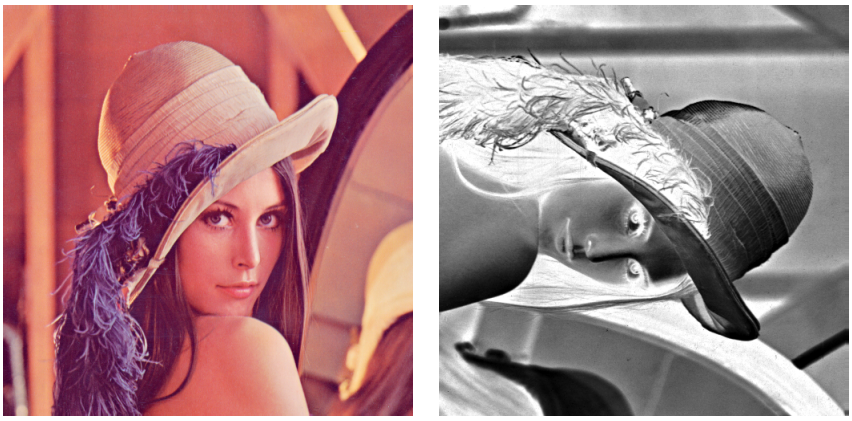

Figure 2: Lena and transformed Lena

3.1 Architecture

This application will be divided into 3 classes (one per file):

- `BitmapExtension`, this static class will contain all the different filters and operations to apply on a given `Bitmap` image,
- `ColorExtension`, this static class will contain atomic operations applied to pixels,
- `MyImageMagick`, this will be the class containing your `Main` method, and the logic of your command line application.

Most (almost all) of the methods you will be asked to implement will be extension methods. We hope you correctly understood the corresponding section in the first part of this subject, if not, please read it again, or consult the official C# documentation [9], or ask your ACDC.

3.2 Basic filters and operations

In this part, you will implement general filters to modify Bitmap images. One or two might seem a bit trickier, but nothing really hard.

All these filters will be extension methods of the `Bitmap` class (this class is `sealed`, so we can not inherit from it). Extension methods need to be static methods, implemented in a static class. The class containing extension methods for `Bitmap` will be `BitmapExtension` (orginal name isn't it ?).

Important: For all this section and the rest of the practical, all extension methods of `Bitmap` have to be defined in the `BitmapExtension` class, and all extension methods of `Color` in `ColorExtension`. Misplaced methods will not be evaluated !

[9] `https://msdn.microsoft.com/en-us/library/bb383977.aspx`

Do this on all of those

Most of the filters you can implement on images are just a repetition of the same modification applied to all of the pixels of a picture. To do so, you will implement a `ForEachPixel` method, this method will be an extension of `Bitmap`, and will take a delegate as an argument. This delegate will be a `Func`, taking a `Color` as argument, and returning the modified color. Here is the content of your `BitmapExtension.cs` after implementing this method:

```
1  // Some usings
2  public static class BitmapExtension
3  {
4    public static void ForEachPixel(this Bitmap image,
5                                     Func<Color, Color> modify)
6    {
7      // Replace each pixel of the picture with
8      // the result of the modification
9    }
10   // Other Bitmap extension methods will be implemented here
11 }
```

Become a smurf

You will here implement the `Invert` method, corresponding to the filter of Figure 1. The inversion of an image is just the inversion of each pixel of it, so you will need to use your `ForEachPixel`. But to use it, you need to know how to invert a `Color`.

Therefore, you will first implement the `Invert` extension method of the `Color` structure. Create your `ColorExtension` static class, and create a static method `Invert`:

```
1  // Some usings
2  public static class ColorExtension
3  {
4    public static void Invert(this Color color)
5    {
6      // FIXME
7    }
8    // Other Color extension methods will be implemented here
9  }
```

Remember here, each channel of color has a value between 0 and 255. To invert a color, each channel must be changed to the complementary of its value on this range: 0 becomes 255, 255 becomes 0, 128 becomes 127.

After implementing the color inversion, you can come back to the image inversion, in `BitmapExtension`:

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

If you correctly use your `ForEachPixel`, and color inversion, this method should not take more than one line. Your `ForEachPixel` must take a delegate (`delegate` or lambda expression), which take a `Color` and returns the invert of it.

Back to the 50's

You are a hipster. Colored picture are too mainstream, you miss the good old days, when pictures were just about brightness and light. Because of that, you will implement a method which applies a grayscale filter on a picture. You will follow exactly the same process as the one presented above for `Invert`. Implement the two following methods in their corresponding classes:

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

The first one must use the second one and the `ForEachPixel` method and has no reason to be longer than one line.

We ask you not to implement an average-based grayscale, but a grayscale which takes into account that the human eye is more easily triggered by green and red lights than blue ones. Use this formula, corresponding to the ITU-R BT.709 standard[10]:

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

The light side vs. the dark side

Because of the evil Palplatine (also known as *Shepard*), you're becoming darker and darker, but sometimes you are able to fight it and come back to the light. To do so, you need to be able to adjust the brightness of yourself.

You will here implement two methods (as done before) to modify the brightness of your picture by modifiying the brightness of each pixel:

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

`delta` is an integer representing the amount of light to add (or substract) to each color. If this value is greater than 255, the resulting picture will be white only, if it is less than -255 it will be black only.

[10] `https://en.wikipedia.org/wiki/Luma_(video)`

Of course, when you add the delta value to the value of a channel, the resulting integer must not be greater than 255 or less than 0. We highly recommend you to implement the following methods in your `ColorExtension` class:

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

They only check if their argument is between the bounds 0 and 255, if not it returns the closest bound. These methods are not mandatory, but will be very useful to avoid righting the same bounds checking in all the rest of the practical. The version taking a `double` is not much more that the one taking an `int`, but it will avoid you to think about the conversion when the result of an operation will be a `double`.

Black is darker, white is brighter

Now, you don't just want to become dark, or bright. We want to emphasize our traits. We both have a light side and a dark side inside us, we want to increase the difference between these sides.

Implement the `Contrast` methods inside their corresponding classes:

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

The relation between these two methods is the same as the methods above, `delta` is supposed to be between -255 and 255 for *normal* result, but other values must work, the resulting image might just look odd.

To compute a contrast change, the first step is to calculate a correction factor which is given by the following formula[11]:

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

Once you computed this factor, to compute the new value of each channel, use the formula:

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

Be careful, the resulting value can be less than 0 or greater than 255, if you implemented it, use correctly your `Restrict256` method.

[11] `http://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-5-contrast-adjustment/`

Gradient map

Here, you will implement a less known filter, the gradient map. This operation replaces each pixel by a value on a gradient, depending on its brightness. In our case, we will represent the gradient with 2 colors, the begin and the end of the gradient. For example, take a look to this picture:

Figure 3: Blue to orange gradient map on Lena

This is the result of the application of a gradient map between blue and orange on the picture of Lena. All the darkest pixels have now a value close to blue, the light ones a value close to orange. To compute the intermediate colors, you have to create a affine function (three actually, one per channel), representing the evolution of the resulting color, when the original pixel color evolves between 0 and 255 (average brightness).

Consider the following theoretical example for a one channel color (you will have to do this 3 times, for each channel in the RGB color model):

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

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

bright is the color matching the white pixels (orange in the Lena example), *dark* is the color matching the black pixels (blue in the example).

Now that you understood how this filter works, implement the two methods, as we did before:

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

As usual, the first one just call the `ForEachPixel` method, all the logic of the transformation is implemented in the second one, transforming `Color` objects.

Transform me!

You will now implement 4 extension methods for `Bitmap`, to transform the picture. You will implement two rotations, and two symmetry transformations:

```
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);
```

The first one and the second one rotate the picture 90°, clockwise and counterclockwise respectively. The last methods apply symmetry on the given image. `SymmetryX` applies a *horizontal* symmetry (up-down), and `SymmetryY` a *vertical* symmetry (left-right). Be careful not to switch the different methods.

Hint: Each of these 4 methods takes only 1 line. Read carefully the MSDN page of the `Bitmap` class[12] to not loose any time on this exercise.

Cover this breast, that I dare not see.

In this exercise, you will learn how to merge two images. The method you will implement will take 2 images and a given opacity, and generate the superposition of the two images. There is nothing difficult here. The only thing you need to know is that you can't just add the pixel values of the two pictures, the resulting picture would be much brighter, and this not what we want. You need to compute the weighted average of the two pictures, to create a realistic transparency.

For this, you will have to implement two methods:

```
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);
```

The argument `int opacity = 50` indicates that the parameter has a default value. If the `opacity` argument is not given to the method, then its value will be 50. This argument is a percentage value, in each of these 2 methods, you must check that the given percentage is valid, correct it otherwise (0 if the value is negative, 100 if the value is two high).

The first method iterates on each pixel of both pictures, call the second method on each of them and replace them. If the two pictures have not the same size, you must throw an `ArgumentException` with an appropriate message.

The `Cover` method for `Color` objects is a weighted average:

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

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

Of course, you must transpose this formula in C# for each of the three color channels.

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

Take a look at this example, we are covering the shot of Lena, with a photograph of fruits[13], with an opacity of 60%. Note that it would be totally equivalent to cover the fruits image with Lena and an opacity of 40%.

Do not try to use `ForEachPixel` in this exercise. The modification to appyl does not depend only on the source image, so this method is not applicable here.

3.3 Gradient generator

Implement the following method:

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

This method draws a gradient from color *left* to color *right* on the given image (overriding existing pixels if any). We ask you a linear, left-to-right, gradient, do not do anything fancier.

Here is an image generated with this method, as an example:

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

[13] `http://www.birpo.com/image/categories/Fresh-Fruits.jpg`

3.4 Palette Extractor

We'll now ask you to implement the following extension method:

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

We recommand you to read the exercise completely before starting it. This method will draw a palette of the n most frequent colors in the input image, splitting the image in n vertical strips of the n most frequent colors from left to right. Here is an example of a palette of the 5 most frequent color of this dear Lena.

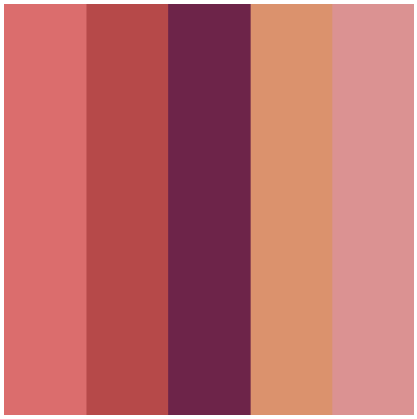

Figure 6: Palette of the 5 most frequent colors of the Lena picture

Each strip is $1/\text{n}^{th}$ of the width of the image. But it is not as simple as taking each pixel's color. Indeed, in a photo, you often have lot of pixel with very similar color, but very small variations, that make them slightly different colors. It would be of course pointless to take them as is : our palette could be basically of one color with almost imperceptible variations. We will then want to approximate the color of our picture to a restricted set of colors. For this practical, we want you to work on a set of 512 colors. To compute these 512 colors, you need to divide each of the 3 channel in eight possible value, between 0 and 255 (0 and 255 being two of these values). Use doubles in your computations for maximum precision, and convert them into integer when you actually create a color.

Once your 512 colors are computed you need a way to approximate the colors of each pixel of your image to one of the set of 512 colors. To do this, you will need to calculate the difference between two colors. Here is a formula to achieve this:

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

You will also need to keep the number of appearance of each of your 512 colors. Using a `Dictionary<Color, int>` object may be a good way to go. When you are done processing your image, you will need to take the n most represented colors (hint : Check MSDN if you used a Dictionary), and draw the corresponding rectangles.

3.5 Command line application

Now is time to make a true program. We will ask you to take arguments from the command line (remember the `string[] args` parameter of your `Main`?). Your program will take at least 3 arguments: the first one will be the path of the input image. The last will be the path of the output image. Between those, you can have at least one, but an unlimited number of options, that corresponds to treatments. The treatments must be executed in sequential order. For instance the following command:

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

Will load the `img/lena.png` file, apply a gray-scale, invert the colors of the gray-scaled image, rotate it, and modify the contrast of the rotated inverted gray-scaled Lena, and then save it in `img/result.png`.

Here is the usage and the list of options your program must recognize:

```
Usage: ./MyImageMagick.Exe source-image {options} destination-image
Options
-b, --brightness VALUE
-c, --contrast VALUE
-m, --cover PATH OPACITY
-d, --gradient COLOR COLOR
-a, --gradient-map COLOR COLOR
-g, --grayscale
-i, --invert
-p, --palette NUMBER
-l, --rotate-left
-r, --rotate-right
-x, --symmetry-x
-y, --symmetry-y
```

As you can see, some options requires to take values. Most of them are number. Some are colors: for these, you are expected to use the method `Color.FromName` that takes a string and returns a corresponding `KnownColor`[14]. If the command line arguments are not correct, display a message on the standard error stream and stop the program: you must not crash because of an not handled exception.

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

4 Boni:

Several boni are possible:

- Blur filter (Gaussian, directional, etc.),
- Convolution matrix,
- Threshold filter,
- Edge-detect filter,
- Meme-like text addition,
- Crop transformation,
- Scale transformation,
- Artistic filters,
- Radius gradient,
- Perlin noise generator,
- Print the palette as a multi-color gradient (rainbow),
- Anything else surprising.

If you add any, do not forget to list them in your `README` file, or they will not be evaluated. Also, these boni must be additional options, and should not affect the behaviour of existing filters.

The code is the law.