# Programación orientada a objetos.

### Letra Obligatorio.

## Introducción

El objetivo de este obligatorio es construir un programa que permita dibujar figuras en pantalla, compuestas por formas básicas. A estas figuras se les podrá aplicar diversas transformaciones en una secuencia dada. Toda la información de lo que se muestra en pantalla va provenir de un archivo de texto de entrada con un formato definido en esta letra.

Para hacerles la tarea un poco más sencilla, la letra del obligatorio se divide en cuatro problemas más simples. Es recomendable que realicen las diferentes partes en orden,

Todo lo mencionado en esta letra deberá ser estar implementado en la entrega final.

**Documentación**

Todo tu programa debe incluir documentación de los métodos y todas las variables de instancia que creas necesario.

**Librería**

No es recomendable modificar la librería propuesta, pero podrás hacerlo siempre y cuando tu programa compile correctamente y cumpla con todo lo requerido por este obligatorio.

**Buenas prácticas**

Recuerda que debes seguir las buenas prácticas de C# al momento de definir variables, nombres de métodos, etc. De no seguir esto, podrás perder puntos en la nota final.

## Parte 1

Mira el siguiente programa compuesto por los archivos Program.cs y Downloader.cs. El programa muestra en la consola el contenido de un recurso -un archivo- de tu disco local pero puedes modificarlo fácilmente para mostrar el contenido de cualquier recurso local o remoto.

¡Prueba armar un proyecto con esos archivos en tu editor favorito y descarga páginas de Internet! Busca en el código de la clase Program las pistas para hacerlo.

Program.cs:

```
using System;
using System.IO;
using System.Reflection;

namespace ExerciseOne
{
    /// <summary>
    /// Pequeño programa para probar el funcionamiento de la clase Downloader.
    /// </summary>
```

```
    public class Program
    {
        /// <summary>
        /// Punto de entrada
        /// </summary>
        public static void Main()
        {
            // Para uniformizar la entrega del obligatorio les pedimos que usen un archivo
            // llamado "archivo.txt" que debe estar en el mismo directorio que el programa.
            // En Visual Studio, pueden agregar un nuevo elemento del tipo archivo de
            // texto y llamarlo "archivo.txt" a su solución; luego cambien la propiedad
            // "CopyToOutputDirectory de ese nuevo elemento a "Copy always": con esto
            // podrán modificar el archivo desde el entorno integrado y asegurarse que al
            // depurar la versión más reciente del archivo se copia al directorio desde donde
            // se ejecuta el programa.

            const String fileName = "archivo.txt";

            String path = Path.Combine(
              Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), fileName);
            UriBuilder builder = new UriBuilder("file", "", 0, path);
            String uri = builder.Uri.ToString();

            // Creamos un nuevo descargador pasándole una ubicación.
            Downloader downloader = new Downloader(uri);

            // Pedimos al descargador que descargue el contenido
            string content;
            content = downloader.Download();

            // Imprimimos el contenido en la consola y esperamos una tecla para terminar
            Console.WriteLine(content);
            Console.ReadKey();
        }
    }
}
```

Downloader.cs:

```
using System;
using System.IO;
using System.Net;
namespace ExerciseOne {
    /// <summary>
    /// Descarga archivos de una ubicación de la forma "http://server/directory/file" o
    /// "file:///drive:/directory/file"
```

```
/// </summary>
public class Downloader {
    private string url;
    /// <summary>
    /// La ubicación de la cual descargar
    /// </summary>
    public string Url { get { return url; } set { url = value;  } }

    /// <summary>
    /// Crea una nueva instancia asignando la ubicación de la cual descargar
    /// </summary>
    /// <param name="url"></param>
    public Downloader(string url) {
        this.url = url;
    }

    /// <summary>
    /// Descarga contenido de la ubicación de la cual descargar
    /// </summary>
    /// <returns>Retorna el contenido descargado</returns>
    public string Download() {
        // Creamos una nueva solicitud para el recurso especificado por la URL recibida
        WebRequest request = WebRequest.Create(url);

        // Asignamos las credenciales predeterminadas por si el servidor las pide
        request.Credentials = CredentialCache.DefaultCredentials;

        // Obtenemos la respuesta
        WebResponse response = request.GetResponse();

        // Obtenemos la stream con el contenido retornado por el servidor
        Stream stream = response.GetResponseStream();

        // Abrimos la stream con un lector para accederla más fácilmente
        StreamReader reader = new StreamReader(stream);

        // Leemos el contenido
        string result = reader.ReadToEnd();

        // Limpiamos cerrando lo que abrimos
        reader.Close();
        stream.Close();
        response.Close();
        return result;
    }
```

```
   }
}
```

Si has visto alguna vez el código HTML de una página web, habrás visto que se organiza mediante *tags*. A grandes rasgos un *tag* tiene la siguiente forma:

```
<nombreTag ([claveAtributo="valorAtributo"] …)>[Contenido]</nombreTag>
```

- `nombreTag` es el **nombre del tag**
- los **atributos** son una lista de tuplas **clave/valor**, de forma que solo pude haber un valor asociado a una clave para un *tag* determinado. Un tag puede tener de 0 a n atributos. Los valores del atributo van siempre entre comillas dobles.
- El **contenido** puede ser cualquier texto.
- Luego del contenido, un tag siempre se cierra mediante una entrada de la forma `</nombreTag>`
- Opcionalmente, si no hay contenido, un tag puede estar cerrado "**en el lugar**" (*in place*) mediante la siguiente sintaxis:

```
<nombreTag ([claveAtributo="valorAtributo"] …) />
```

- Se permiten cualquier cantidad de espacios antes del nombre del *tag*, entre el nombre del tag y el final del tag o primer atributo, entre un atributo y el símbolo de =, entre el símbolo de = y el valor del atributo y entre éste y el final del tag o el próximo atributo.
- Se dice que el tag esta **mal formado** cuando no respeta las reglas de sintaxis, por ejemplo: no se cierra, tiene múltiples atributos con la misma clave, etc.

**Ejemplos**

**`<b>Negrita</b>`**
Un tag de nombre b, sin atributos, cuyo contenido es `Negrita`.

**`<br/>`**
Un tag de nombre `br`, sin atributos, sin contenido.

**`<font color="red">Rojo</font>`**
Un tag de nombre font, con un atributo con clave color y valor red, con contenido Rojo

**`<input type="text" name="nombre"/>`**
Un tag de nombre input, con un atributo con clave type y valor text y un atributo name con valor nombre.

Los siguientes son tags mal formados:

**`<input type="text" type="text"/>`**
Mal formado, clave de atributo repetido.

**`<input type="text" name="nombre">`**
Mal formado, el tag no se cierra.

**</input>**

Mal formado, el tag no se abre.

**<type="text" name="nombre"/>**

Mal formado, no tiene nombre.

Tu primera tarea es construir un programa que lea un archivo de texto e imprima el nombre de cada *tag* y sus atributos. Los *tags* deben ser impresos en el orden en el que aparecen en el archivo. El orden en que se impriman los atributos no es importante. Tampoco es necesario, para esta primera instancia, que evalúes si el *tag* se cierra correctamente o no.

Por ejemplo, si el archivo contiene:

```
<html><body><font color="blue" size="3">Ingrese su nombre </font><input
type="text" name="nombre" maxlength="8"/><br/><font size="2">Máximo 8
caracteres</font></body></html>
```

(que, por cierto, es contenido HTML "válido" que puedes abrir con tu navegador)

El programa debe imprimir por consola:

```
html
body
font
color=blue
size=3
input
type=text
name=nombre
maxlength=8
br
font
size=2
```

Tu programa puede imprimir lo mismo incluso si el archivo contiene *tags* mal formados porque no se cierran, como por ejemplo:

```
<font color="blue" size="3">Ingrese su nombre <input type="text" name="nombre"
maxlength="8"><br><font size="2">Máximo 8 caracteres
```

Ésto solo para simplificar el problema. Por supuesto que si deseas chequear que todos los *tags* deben estar bien formados, adelante.

Para resolver el problema, pueden serte útiles las clases del namespace System.Collections, en especial Hashtable y ArrayList. Usa la documentación de .NET Framework SDK para saber como utilizarlas y que colaboración puedes pedir a instancias de éstas clases. Además, existe en .NET una librería para el manejo de documentos XML que puede serte útil.

# Parte 2

Un cuadrado, un triangulo rectángulo y un rectángulo son todos polígonos. Es posible calcular para cada uno de ellos, mediante matemática básica, su área y su perímetro.

a. Programen clases que permitan representar triángulos rectángulos, cuadrados y rectángulos. Traten de identificar un tipo común a todos ellos y apliquen lo que saben de tipos e interfaces para representar este tipo común en su código.

b. A partir de un archivo XML que describe una serie de triángulos rectángulos, cuadrados y rectángulos, impriman el perímetro y el área de cada uno de ellos.

El archivo XML contiene tres tipos de tags distintos:

Tag de nombre **triangle**: Representa un triangulo, tiene dos atributos base y height cuyos valores son números enteros que representan el largo de la base y la altura del triángulo respectivamente. No tiene contenido.

*Ejemplo:*

<triangle base=”10” height=”20”/>

Es un triángulo de base 10 y altura 20, por lo tanto su área es de 10*20/2=100, y su perímetro es de $10+20+\sqrt{10^2+20^2}= 52,3$.

Tag de nombre **square**: Representa un cuadrado, tiene un atributo length cuyo valor es un número entero que representa el largo del lado del cuadrado. No tiene contenido.

*Ejemplo:*

<square length=”20”/>

Es un cuadrado de lado 20, por lo tanto su área es de 20*20=400, y su perímetro es de 20*4=80.

Tag de nombre **rectangle**: Representa un rectángulo, tiene dos atributos width y breadth cuyos valores son números enteros que representan el ancho y largo del rectángulo respectivamente. No tiene contenido.

*Ejemplo:*

<rectangle width=”20” breadth=”30”/>

Es un rectángulo de ancho 20 y largo 30, por lo tanto su área es de 20*30=600, y su perímetro es de 20*2+30*2=100.

Adicionalmente cada tag tiene un atributo de clave id que define un identificador para la figura.

Por ejemplo:

<rectangle id=”rectangulo1” width=”20” breadth=”30”/>

Dentro de un archivo de figuras, cada figura debe tener un identificador único distinto al del resto de las figuras.

Su programa debe imprimir el identificador de cada figura contenida en el archivo seguido por el perímetro y el área de la misma.

Por ejemplo, si el archivo contiene:

<rectangle id=”rectangulo1” width=”10” breadth=”30”/>
<triangle id=”triangulo1” base=”10” height=”20”/>
<square id=”cuadrado1” length=”30”/>
<rectangle id=”rectangulo2” width=”20” breadth=”30”/>
<square id=”cuadrado2” length=”40”/>

El programa debe imprimir:

rectangulo1 área=300 perímetro=80
triangulo1 área=100 perímetro=52,3
cuadrado1 área=900 perímetro=120
rectangulo2 área=600 perímetro=100
cuadrado2 área=1600 perímetro=160

Realiza una buena distribución de responsabilidades. Ten en cuenta que el día de mañana pueden haber nuevas figuras para las cuales quiera saber el área y perímetro y el programa no debería cambiar significativamente.

# Parte 3

Ahora ya podemos comenzar a dibujar las figuras en la pantalla.

Para eso cuentas con una clase provista por nosotros que implementa `IPainter` siempre y cuando le proveas un objeto de una clase que implemente la siguiente interfaz `IShape`:

```
using System;

namespace Drawing
{
	/// <summary>
	/// Un punto de un espacio de dos dimensiones (X e Y).
	/// </summary>
	interface IPoint
	{
		/// <summary>
		/// Retorna la coordenada X.
		/// </summary>
		int X { get; }
		/// <summary>
		/// Retorna la coordenada Y.
		/// </summary>
		int Y { get; }
	}

	/// <summary>
	/// Una figura esta compuesta por todos los vértices de la figura.
	/// </summary>
	interface IShape
	{
	  /// <summary>
	  /// Retorna los vértices de la figura.
	  /// </summary>
	  IPoint[] Points { get; }
	}
	/// <summary>
	/// Pinta figuras en pantalla.
	/// </summary>

	interface IPainter
	{
		/// <summary>
		/// Pinta la figura en la pantalla.
		/// </summary>
		/// <param name="shape">La figura a pintar.</param>
		/// <param name="color">El color de la figura.</param>
		void paint (IShape shape, Color color);
	}
}
```

Para esto necesitas modificar tu programa de la parte 2 para permitir especificar el punto de origen de tus figuras y calcular los vértices de cada una.

Para eso, definimos dos atributos adicionales para cada tag, llamados X e Y, que definen el punto de origen para la figura en las coordenadas X e Y.

Por ejemplo,

<rectangle id="rectangulo1" X="1" y="1" width="10" breadth="30 "/>

Define un rectángulo cuyo punto de origen es el punto (1,1), tiene 10 puntos de ancho y 30 de largo.

Las reglas para calcular los vértices para cada figura son las siguientes:

**square**

Si posición de origen = (x,y), los vértices de la figura son: (x,y), (x + length - 1, y), (x + length - 1, y + length - 1), (x, y + length -1).

**rectangle**

Si posición de origen = (x,y), los vértices de la figura son: (x,y), (x + breadth - 1, y), (x + breadth - 1, y + width - 1), (x, y + width - 1).

**triangle**

Si posición de origen = (x,y), los vértices de la figura son: (x,y), (x + base - 1, y), (x + base - 1, y + height - 1).

Los -1 son debido a que las dimensiones especifican la cantidad de puntos y el origen ya es un punto. Por lo tanto, un cuadrado de origen (x,y) de un punto de lado es simplemente un punto, de dos puntos de lado un cuadrado (x,y),(x+1,y),(x+1,y+1),(x,y+1).

El tamaño de la grilla donde se puede dibujar es ilimitado (o mejor dicho el tamaño máximo de un entero). Sin embargo, la implementación de IPainter que les daremos solo muestra una grilla de 100x100 puntos (un cuadrado (0,0), (99,0), (99,99), (0,99)).

Estos vértices son los que debe retornar la propiedad IPoints para las distintas figuras. Deben asegurarse de que el orden sea idéntico al especificado.

El punto (0,0) se encuentra en la esquina inferior izquierda de la pantalla.

**Color**

Color es una clase definida en System.Drawing. Pose un método estático con la siguiente firma que permite obtener una instancia de la clase a partir del nombre del color:

```
public static Color FromName(string name);
```

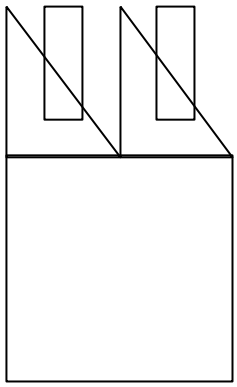
Por ejemplo, para obtener el color rojo pueden escribir:

```
Color red = Color.FromName("red");
```

# Parte 4

Ahora ya estamos en condiciones de comenzar a formar figuras compuestas de otras figuras. Por ejemplo, si quisiera dibujar en pantalla una fábrica de papel podría, con algo de ingenuidad, dibujar un cuadrado con dos triángulos rectángulos sobre el mismo y un par de chimeneas.

Ilustración 1: Fábrica de papel

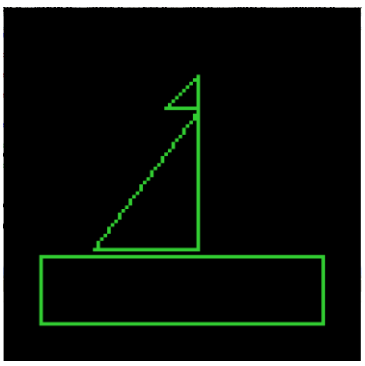


Modifica tu programa para que permita definir dibujos que se componen de figuras. Para esto extendemos nuestro archivo de configuración para que permita definir dibujos compuestos de otras figuras:

```
<rectangle id="casco" x="10" y="10" width="20" breadth="80"/>
<triangle id="mayor" x="25" y="31" base="30" height="40"/>
<triangle id="banderin" x="45" y="71" base="10" height="10"/>
<drawing id="barco" components="casco mayor banderin"/>
```

Como ven el dibujo se define mediante un tag de nombre drawing, tiene un identificador que permite referenciarlo en otras partes del archivo, y posee un atributo con clave components cuyo valor es una cadena con los identificadores de las figuras que lo componen separados por espacio

Al dibujar esta dibujo debería aparecer un estilizado barco como el siguiente:.



¡Ten cuidado en como defines un dibujo! Puede parecerte natural hacer que un dibujo sea una IShape, de forma de poder pasarla a un pintor para que la pinte; pero si lo haces así estás en un gran problema. La interfaz IShape tiene una propiedad que retorna los vértices de la figura y en un

dibujo identificarlos no es nada sencillo (no te apresures, ya podrás atacar estos problemas en el curso de Computación Gráfica). Las figuras podrían solaparse y el pintor también asume que los puntos definen una figura cerrada (que no es el caso, como se ve en el ejemplo). Un pintor nunca va a poder pintar dibujos, pero un dibujo podría indicarle al pintor como pintarlo.

Para que sea más sencillo, pueden asumir que las figuras deben estar definidas antes que se defina el dibujo. Si al intentar definir un dibujo no encuentran uno de sus componentes pueden levantar una excepción e informarle al usuario del error de configuración.

### 4.2.

En el ejemplo visto en la parte anterior, es claro que queremos pintar solo el dibujo, pero no las figuras individuales (¡en otro caso estaríamos pintando dos veces lo mismo!). Para esto, definiremos una *acción* de pintar. Cuando en el archivo de configuración aparece una acción de pintar, se debe pintar la figura referenciada por esa acción. Por ejemplo:

```
<rectangle id="rectangulo1" x="60" y="20" width="5" breadth="10"/>

<paint figure="rectangulo1" color="red"/>
```

Esto define un rectángulo (que por defecto no se pinta) y luego una acción de pintar que pinta ese rectángulo. Una figura solo se pinta en pantalla si aparece una acción de pintar. Si no hay acciones de pintar, la pantalla aparecerá en negro.

El nombre del color con el cuál pintar la figura se define cuando aparece la acción de pintar, por ejemplo:

```
<paint figure="barco" color="red"/>
```

Implica pintar la figura barco con el color identificado por el nombre “red” (acertaron, rojo).

Nuevamente, ten en cuenta que un pintor, tal como fue definido, sabe pintar figuras, no dibujos. Piensa quién será el encargado de pintar los dibujos.

Ten en cuenta que en el futuro se podrían definir nuevas acciones que se apliquen sobre una figura. Además, en el futuro también vamos a querer definir dibujos que se compongan de otros dibujos.

### 4.3.

Implementa transformaciones geométricas básicas para tus figuras.

Como mínimo, define la simetría axial de forma que puedas generar espejos de tus figuras.

Tu archivo de configuración (¡o ya podríamos llamarle lenguaje de definición de dibujos!) debe permitir definir estas transformaciones.

La sintáxis para la simetría axial es la siguiente:

```
<simetry-x coordinate="5" figure="barco"/>
```

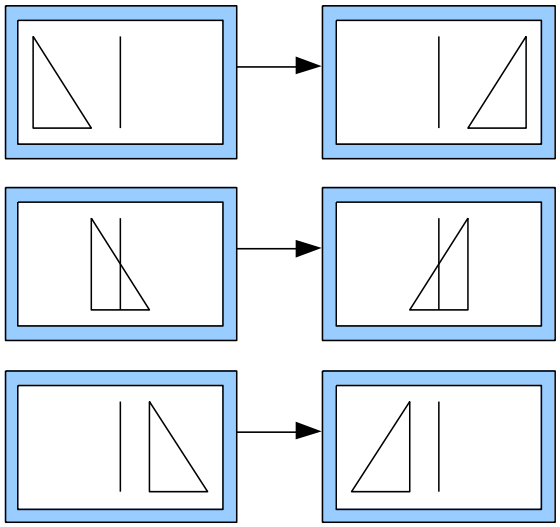
Una simetría axial sobre el eje de las X = 5.

```
<simetry-y coordinate="7" figure="barco"/>
```

Una simetría axial sobre el eje de las Y = 7.

Los siguientes son ejemplos de aplicar una simetría axial con eje X.

Ilustración 3: Simetrías axiales de eje X

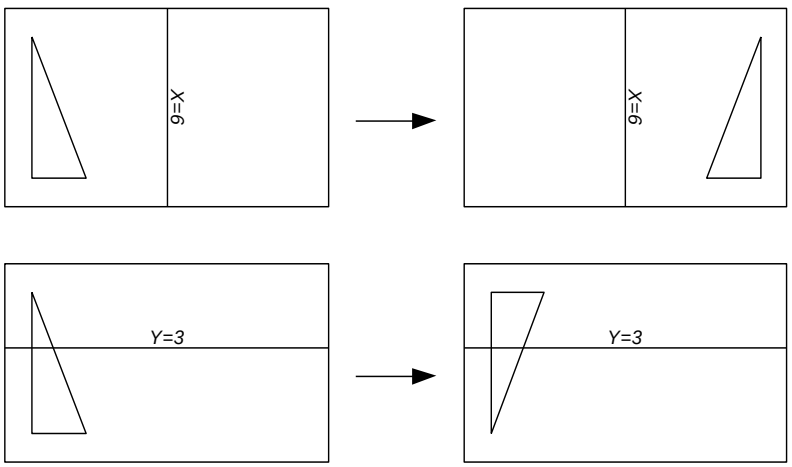


Aplicando simetrías axiales a un tríangulo de coordenadas (1,1), (1,5), (3, 5) obtenemos:

Para X=6, (11, 1), (11,5), (9,5) o en forma general, para X = Sx, ( (Sx-x) + Sx, y).

Para Y=3, (1, 5), (1,1), (3,1) o en forma general, para Y = Sy, (x, (Sy – y) + Sy).

Ilustración 4: Siemetrías axiales de ejemplo




Es válido que alguna coordenada quede en un valor negativo; quien pinta la figura (el IPainter) se encargará de decidir que hacer al respecto.

Pueden aplicarse dos transformaciones a la misma figura, por ejemplo:

<simetry-x coordinate="5" figure="barco"/>

<simetry-y coordinate="7" figure="barco"/>

Implica que debes primero hacer una simetría axial de eje X=Y de la figura barco y luego una de eje Y=7 para la figura barco que acabas de transformar.

Ten en cuenta que al momento de aplicar una transformación la figura debe conservar su identidad. Es decir, la transformación debe hacerse sobre la propia figura en lugar de realizar una copia de la misma transformada. Presta atención a que todas las transformaciones mencionadas hasta el momento se definen en base a un punto y no a una figura (es decir, transformar una figura es lo mismo que transformar cada uno de sus puntos).

Para simplificar, las transformaciones están definidas después de las figuras, por lo cual es un error que la transformación haga referencia a una figura que no está definida.

Será importante el orden en el cual se apliquen las acciones sobre las figuras, por ejemplo:

<paint figure="barco"/>

<simetry-x coordinate="5" figure="barco"/>

<paint figure="barco"/>

<simetry-y coordinate="7" figure="barco"/>

Va a pintar la figura barco sin transformaciones y después de realizar la primera transformación, pero no después de la segunda.

## Obligatorio.

Si siguieron la letra en orden hasta aquí, esta última parte debería ser muy simple de implementar. Para la entrega final, vamos a agregar algunas funcionalidades extra al programa.

## Dibujos compuestos de otros dibujos y figuras

Actualmente, su programa permite definir dibujos que se componen de otras figuras. Para la entrega final queremos dar otro paso y permitir que los dibujos se compongan de otros dibujos y de otras figuras.

El siguiente es un ejemplo de un dibujo que se compone de otros dibujos y de un dibujo que se compone tanto de dibujos como de figuras:

```
<rectangle id="casco" x="10" y="10" width="20" breadth="80"/>
<triangle id="mayor" x="25" y="31" base="30" height="40"/>
<triangle id="banderin" x="45" y="71" base="10" height="10"/>
<drawing id="velas" components="mayor banderin"/>
<drawing id="base" components="casco"/>
<drawing id="barco" components="base velas"/>
<square id="marco" x="5" y="5" length="90"/>
<drawing id="barco_en_marco" components="barco marco"/>
```

En este caso, el dibujo barco se compone de los dibujos base y velas, y el dibujo `barco_en_marco` se compone del dibujo `barco` y la figura `marco`.

Piensen si pueden evitar el contemplar el caso de dibujo compuesto de dibujos y figuras, por ejemplo asumiendo que una figura es un dibujo con un único elemento.

Como en los ejercicios previos, un dibujo siempre hace referencias a dibujos o figuras definidos previamente. Es decir, cuando aparece el tag `drawing` es necesario que todos sus componenentes ya hayan sido definidos. En otro caso, pueden levantar una excepción.

## Aplicar transformaciones a los dibujos

Hasta el momento las transformaciones se aplicaban solo a las figuras. Para el obligatorio queremos que las transformaciones se apliquen tanto a las figuras como a los dibujos.

Por ejemplo:

```
<drawing id="barco" components="base velas"/>
<simetry-y coordinate="7" figure="barco"/>
```

Realiza una simetría de eje Y = 7 de la figura barco. El transformar un dibujo equivale a transformar cada uno de sus componentes.

Se puede transformar cualquier tipo de dibujo. ya sea un dibujo compuestos por dibujos, figuras o ambos.

## Nuevas Transformaciones

***Traslación:***

La traslación desplaza los puntos de una figura a lo largo del eje X y del eje Y tantas unidades como

es indicado.

```
<translation x="5" y="5" figure="barco"/>
```

Define una traslación de 5 unidades sobre el eje de las X y de 5 sobre el eje de las Y.

La forma general de obtener el punto trasladado para una traslación de Tx unidades sobre X y Ty unidades sobre Y es (x + Tx, y + Ty).

También se pueden usar unidades negativas para mover en otro sentido:

```
<translation x="-5" y="-5" figure="barco"/>
```

***Simetría Central:***

Una simetría central se define en base a un centro de simetría:

```
<simetry x="5" y="7" figure="barco"/>
```

Es equivalente a realizar una simetría axial de eje X con X = 5 y otra simetría axial de eje Y con Y = 7.

***Rotación:***

Una rotación se define en base al punto de rotación y al ángulo:

```
<rotation x="5" y="7" angle="45" figure="barco"/>
```

El ángulo se expresa en un número entero de grados. La forma general de hacer una rotación es:

```
double RADIAN_RATIO = (Math.PI / 180); //Factor para transformar a radianes

double angle = rotationAngle * RADIAN_RATIO; //Lo transformamos a radianes

double sin = Math.Sin(angle);
double cos = Math.Cos(angle);

int x = point.X - origin.X;
int y = point.Y - origin.Y;
int newX = ((int) Math.Round((x * cos - y * sin))) + origin.X,
int newY = ((int) Math.Round((x * sin + y * cos))) + origin.Y);
```

Donde point es el punto inicial que quiero rotar, origin el punto de origen (en nuestro ejemplo (5,7)), y rotationAngle el valor del ángulo (45).

El ángulo puede ser negativo, en cuyo caso la rotación es en el otro sentido.

## Nuevas Acciones

Deberán poder definir acciones adicionales.

***Pausar***

La acción de pausar detiene el programa durante el tiempo especificado en milisegundos.

```
<pause millis="500"/>
```

En este caso el programa debe detenerse durante 500 milisegundos. El uso de la acción de pausar permite que el programa se detenga entre, por ejemplo, dos acciones de pintar:

```
<paint figure="cuadrado-1" color="red"/>
<pause millis="500"/>
```

```
<paint figure="cuadrado-2" color="green"/>
```

En este caso se pinta el cuadrado 1, se pausa durante 500 milisegundos, y luego se pinta el cuadrado 2.

## Nuevas formas de ejecutar las acciones

Junto a la nueva, renovada y moderna biblioteca de dibujo que les proveemos, se incluye una interface llamada IExecutor, declarada de la siguiente forma (se excluye la documentación por temas de espacio):

```
namespace Drawing
{
   public interface IExecutor
   {
      void AddAction(IAction action);
      void Execute();
   }
}
```

Un ejecutor encola acciones para ser ejecutadas y las ejecuta en forma secuencial, en el orden en que fueron encoladas.

Las acciones que se encolan, son objetos instancias de IAction, una interfaz declarada de la siguiente forma:

```
namespace Drawing
{
   public interface IAction
   {
      void Perform();
   }
}
```

Las acciones deben tener todo lo necesario para ejecutarse cuando se llama a su método Perform() (ya que como se ve, no recibe ningún parámetro). Por ejemplo, si la acción debe pintar, necesita tener un pintor y la figura o dibujo a pintar, en cambio si la acción debe transformar, debe tener la transformación y la figura o el dibujo a transformar (o saber cual es el mensaje a enviar para transformarlo).

Este cambio implica que a partir de éste momento no deben ejecutar las acciones en cuanto las leen, sino que deben encolarlas en un ejecutor y llamar a la operación de ejecutar al final del programa.

### ¡Atención!

Nuestra implementación de ejecutor hace algunas cosas más para simplificar su vida (o no), tal como ejecutar el `Application.Run()` que ustedes llamaban manualmente hasta el momento. Por lo tanto su programa no debe terminar ahora con un `Application.Run`() si no con un `executor.Execute()` o algo similar a ésto.

Por lo tanto, su programa se resume a interpretar el archivo, crear las acciones de forma adecuada, encolarlas y pedirle al ejecutor que las ejecute. Claro que decirlo siempre es más fácil que hacerlo....