LUG MII U4 2018

Lenguajes De Última Generación (Universidad Abierta Interamericana)

Carrera: **Analista Programador**

Lenguaje de última generación

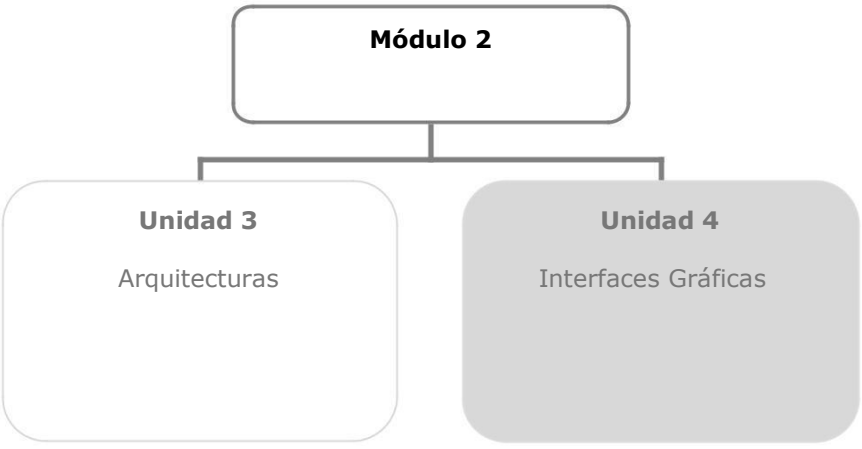
Módulo II

Utilizar técnicas avanzadas de programación

Unidad 4

Interfaces gráficas

Docente y autor de contenidos: Ing. Mauricio Prinzo



Presentación

En esta unidad recordaremos algunos conceptos de los lenguajes de programación que son pilares fundamentales para el desarrollo de aplicaciones empresariales.

Esperamos que usted, a través del estudio de esta unidad, adquiera capacidad para:

Acceder rápidamente a una base de datos relacional
Profundizar su comprensión acerca de las ventajas de trabajar en modo conectado o desconectado
Interpretar y utilizar las estructuras de datos utilizadas en un lenguaje de programación
Trabajar con estructuras planas de datos que faciliten el intercambio de datos

A continuación, le presentamos un detalle de los contenidos y actividades que integran esta unidad. Usted deberá ir avanzando en el estudio y profundización de los diferentes temas, realizando las lecturas requeridas y elaborando las actividades propuestas, algunas de desarrollo individual y otras para resolver en colaboración con otros estudiantes y con su profesor tutor.

Contenidos y Actividades

2.4. Formas con relleno

Trabajo colaborativo/Foro

Foro: Interfaces gráficas

Lectura Requerida

Booch, Grady. **Análisis y diseño orientado a objetos**. Edición número 2. Pearson Educación. 1996. Capítulo I al III.

Cierre de la Unidad

EVALUACIÓN PARCIAL

Propuesta para la Integración del Módulo II
Utilizar técnicas avanzadas de programación

Para el estudio de estos contenidos usted deberá consultar la bibliografía que aquí se menciona:

BIBLIOGRAFÍA OBLIGATORIA

Booch, Grady. Análisis y diseño orientado a objetos. Edición número 2. Pearson Educación. 1996. Capítulo I al III.

Links a temas de interés

http://Biblioteca.vaneduc.edu.ar/

El portal de la biblioteca universitaria con material y links de interés, además del asesoramiento de un referencista especializado.

Lo/a invitamos ahora a comenzar con el estudio de los contenidos que conforman esta unidad.

1. Introducción

El nuevo FrameWork de Microsoft ofrece innumerables ventajas para trabajar con gráficos vectoriales en nuestras aplicaciones.

Aun cuando no es una práctica muy frecuenta en los sistemas comerciales, veremos a continuación un detalle de todas las funciones ofrecidas para su desarrollo y dejaremos bajo su dominio la búsqueda y hallazgo de las posibilidades de uso de cada una.

Para acompañar el estudio de los contenidos que presentaremos, le proponemos la lectura del siguiente texto.

1.1. Qué es GDI - Graphics Device Interface

Es una interface de dispositivo gráfico que aporta las funcionalidades necesarias para que nuestra aplicación se comunique con cualquier dispositivo gráfico de salida conectado a nuestro ordenador, por ejemplo, la pantalla o el monitor.

Generalmente está vinculado a una API (Interfaz de programación de aplicación) que se encuentra en todos los sistemas operativos. Las APIs no son más que una serie de servicios y llamadas al sistema Operativo pero vista desde nuestra aplicación es un conjunto de funciones disponibles para aprovechar al máximo las funciones de nuestro SO.

Evoluciona de GDI (GDI+) es un conjunto de servicios de Windows XP o superior que presenta un conjunto de servicios que nos permite interactuar con dispositivos gráficos como pantallas.

Si lo trabajamos desde el Framework NET, nos encontraremos con la necesidad de importar funciones agrupadas en la siguiente lista de espacios de nombre:

- **System.Drawing**
- **System.Drawing.Drawing2D**
- **System.Drawing.Imaging**
- **System.Drawing.Text**

- **System.Drawing.Printing**

Estos grupos facilitan el uso de estructuras graficas como líneas, polígonos, entre otras, con una practicidad y una facilidad increíble.

La clase **Graphics** encapsula un espacio de dibujo GDI+ y es una clase que no puede heredarse. Esta clase es la base de la funcionalidad proporcionada por GDI+.

Proporciona métodos para dibujar estructuras gráficas en la pantalla (líneas, curvas, figuras, imágenes y texto) y se puede obtener llamando a un objeto **control.CreateGraphics**. Siendo **control** el contenedor.

Un ejemplo de código, si consideramos el formulario como contenedor, se representaría en unas pocas líneas:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{      Graphics g = this.CreateGraphics();
  }
```

GDI+ se divide en tres categorías básicas:

- Gráficos Vectoriales 2D
- Imágenes
- Tipografía

A continuación, nos detendremos en la primera.

1.2. La clase Graphics

Esta clase, encapsula un espacio de dibujo GDI+ y es una clase que no puede heredarse. Proporciona métodos para dibujar estructuras gráficas en la pantalla y se puede obtener llamando a un objeto control.CreateGraphics. Siendo control el contenedor.

Un ejemplo de código, si consideramos el formulario como contenedor, se representaría en unas pocas líneas que se muestra a continuación:

```
private void Form1_Paint(object sender, PaintEventArgs e)
    {      Graphics g = this.CreateGraphics();
```

Utilizando la clase mencionada se pueden dibujar Líneas, Curvas, Figuras, Imágenes y Texto.

Tener en cuenta el evento Paint del Form en el FormDesigner

```
this.Paint += new System.Windows.Forms.PaintEventHandler(this.Form1_Paint);
```

2. Gráficos Vectoriales 2D

Las API de GDI+ son una evolución natural de las funciones de GDI y, por esa razón, pueden ayudarnos a trabajar con imágenes 2D.

Una imagen 2D es aquella que se enmarca en el espacio determinado por la conjunción de los ejes cartesianos X e Y. Esto quiere decir que toda figura 2D tiene coordenadas X, Y para su ubicación. En general, disponen de un valor para representar el ancho y otro, para representar el alto.

2.1. Los Objetos Gráficos

Todos los ejercicios que desarrollaremos comenzarán de la misma forma, con la creación de un lienzo por decirlo de alguna manera. Todas las figuras se crearán sobre un espacio gráfico que dependerá del *contenedor*.

```
Graphics g = this.CreateGraphics();
```

Esta variable de nombre *g* es una instancia real de la clase **Graphics** del contexto donde se encuentra. Y es esta misma clase la que nos permitirá dominar el entorno gráfico.

Uno de los efectos que usaremos en primer lugar es la posibilidad de emular la orden de **limpiar todo el espacio gráfico.** En realidad, limpiar no sería la forma correcta de plantearlo pero emula perfectamente el efecto.

La clase, nos permite pintar todo el lienzo con un color a elección, tal como se muestra en este código:

```
g.Clear(Color.White);
```

Color es una instancia que representa los colores disponibles.

2.2. Líneas, Rectángulos, Polígonos, Elipses y arcos

La lista de estructuras gráficas disponibles es algo limitada pero representa las figuras tradicionales. Partiendo de ellas, podremos generar otras figuras basadas en la transformación o en la complementación de las figuras básicas.

Todas las figuras tienen un proceso similar cambiando solo el resultado, es decir, el tipo de figura.

A continuación, veremos cada una de las sentencias que nos permiten crear estas figuras.

Tenga en cuenta los siguientes aspectos al crear una figura:

1. Cada método deberá pasar un objeto **Pens** como primer argumento y un conjunto de coordenadas o un rectángulo limítrofe en los argumentos restantes.
2. El objeto **Pens** se deberá pasar en cada llamada.
3. Los objetos gráficos no recuerdan cual fue el último punto dibujado.
4. Los objetos gráficos carecen de estado.

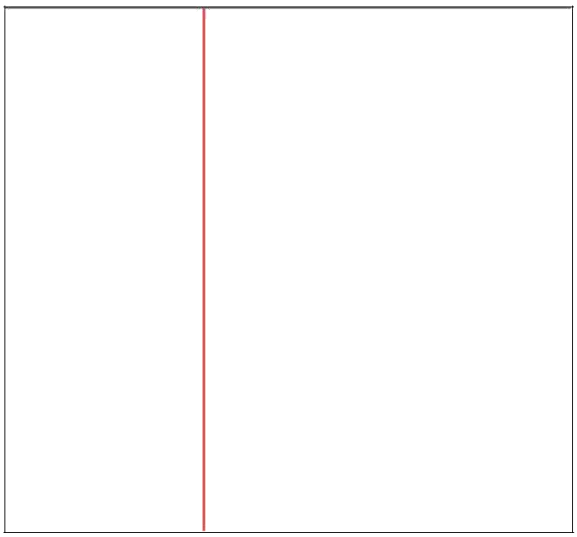
Línea

Todas las líneas necesitan dos grupos de valores, las coordenadas (X, Y) del punto de inicio y las coordenadas (X,Y) del punto de fin.

Una línea es un conjunto sucesivo de puntos que se dibujan uno al lado del otro.

Dibujaremos a modo de ejemplo una línea roja en el formulario, en el evento Paint:

```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);
g.DrawLine(Pens.Red, 100, 0, 100, 300);
```



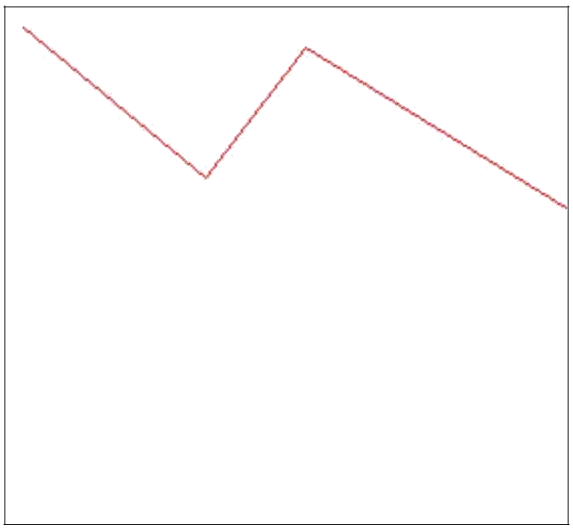
Siguiendo con la misma figura, podemos no sólo dibujar una línea sino agruparlas. Es decir, concatenar las líneas entre sí.

Para poder realizar esto, debemos crear una matriz de objetos de tipo **Point**. Un objeto **Point** es aquel que tiene dos valores, un valor X y un valor Y.

```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);

Point[]  Puntos = {new Point(9, 10), new Point(100, 85), new
Point(150, 20), new Point(280, 100)};

g.DrawLines(Pens.Red, Puntos);
```

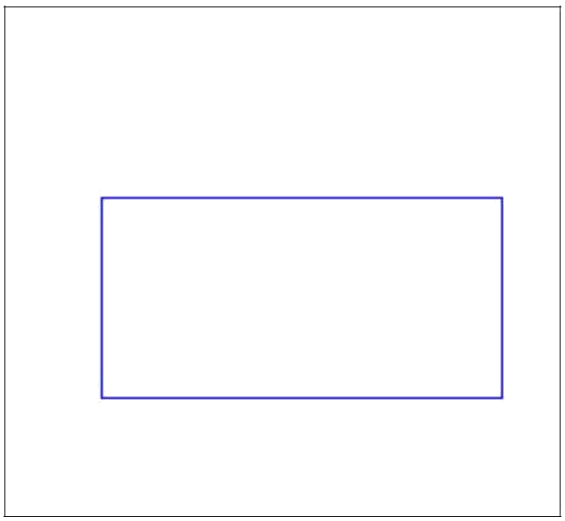


Rectángulo

El método **Drawrectangle** necesita dos argumentos, el objeto *Pens* que define el tipo y color de línea y los valores del rectángulo que se va a dibujar. Este último valor es un tipo **rectangle** que tiene 4 argumentos, las coordinadas (x,y) más el ancho y el alto del rectángulo.

Un rectángulo es una figura donde es necesario establecer el alto y el ancho. Si el alto y el ancho son idénticos, estamos hablando de un cuadrado.

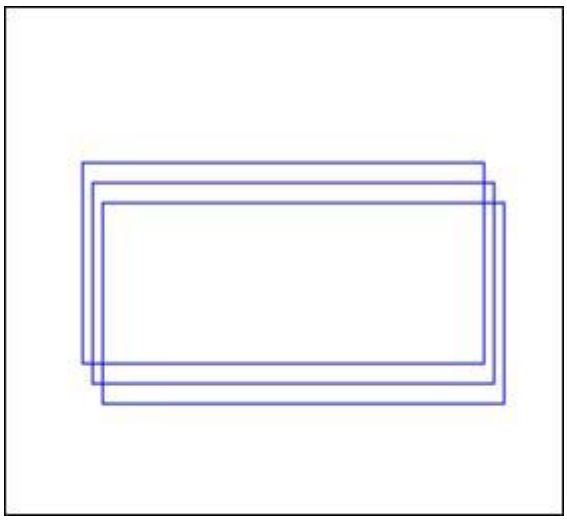
```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);
        Rectangle rect = new Rectangle (50, 100, 200, 100);
        g.DrawRectangle(Pens.Blue, rect);
```



Tal como ocurría con las líneas tenemos el método **DrawRectangles** que nos permite dibujar varios rectángulos. De igual modo creamos una matriz de rectángulos y luego ejecutamos el método.

```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);
Rectangle[] ListaRectangulos = {new Rectangle(50, 100, 200,
100), new Rectangle(45, 90, 200, 100), new Rectangle(40, 80, 200, 100)};

g.DrawRectangles(Pens.Blue, ListaRectangulos);
```



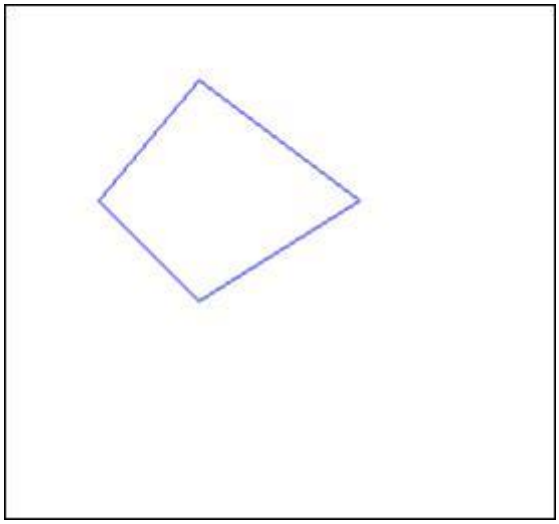
Polígonos

En el caso de nuestro código, la diferencia entre dibujar varias líneas y dibujar un polígono es que para este último haremos coincidir el último punto con el primero.

Son figuras plana formada por una secuencia finita de de segmentos rectos consecutivos y no alineados.

```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);
Point []  Puntos = {new Point(100, 40), new Point(180, 100), new
Point(100, 150), new Point(50, 100)};

g.DrawPolygon(Pens.Blue,Puntos);
```



Elipse

Básicamente para dibujar un elipse, debemos expresar dos valores en sus argumentos: el objeto **pens** para definir el color y los valores del rectángulo que definen el área de contención.

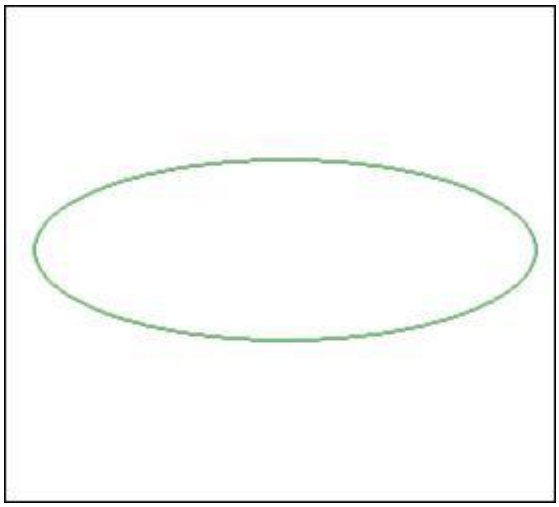
Es una curva cerrada y plana donde la distancia desde su centro a los extremos puede ser diferente o igual en cuyo caso dibujamos un círculo.

La elipse se dibuja automáticamente dentro del área definida por el rectángulo.

```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);

Rectangle rect  = new Rectangle(20, 80, 250, 90);

g.DrawEllipse(Pens.Green, rect);
```



En el siguiente ejemplo dibujamos un círculo de color verde, noten que el alto y el ancho del rectángulo son iguales

```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);
)

Rectangle rect = new Rectangle(80, 80, 90, 90);

g.DrawEllipse(Pens.Green, rect);
```

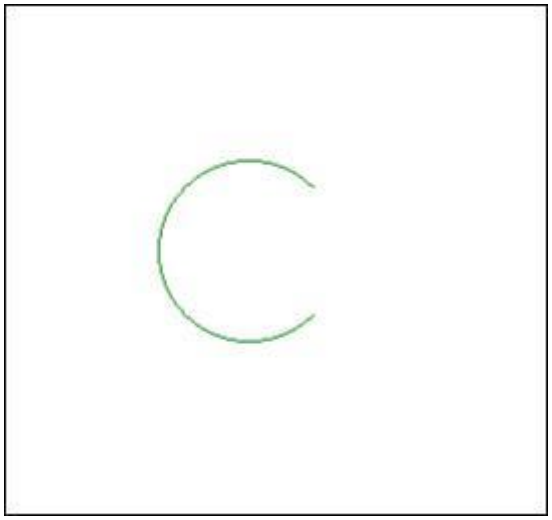
Arcos

Es similar al rectángulo pero debemos pasarle dos argumentos más, que son dos ángulos

Dibuja una curva. Podríamos considerar al arco como una elipse incompleta.

```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);

Rectangle rect  = new Rectangle(80, 80, 90, 90);
Single anguloinicio = 45.0F;
Single angulofin = 270.0F;
g.DrawArc(Pens.Green, rect, anguloinicio, angulofin);
```



2.3. Curvas: Splines de Bezier y Cardinales

Los **Cardinales** representan curvas de un material flexible con un grado de tensión llamado *flexibilidad*.

Se pueden crear pasándole una serie de puntos fijados en el plano X e Y. Por defecto, la tensión está pautada en 0,5.

Ejemplo de uso[1]

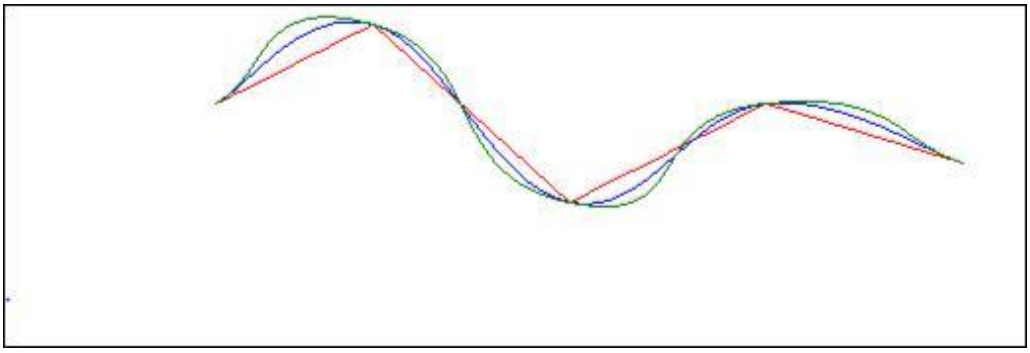
```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);

Point[] Puntos = {new Point(20, 50), new Point(100, 10), new
Point(200, 100), new Point(300, 50), new Point(400, 80)};

e.Graphics.DrawCurve(Pens.Red, points, 0.0F);
```

1 http://msdn.microsoft.com/es-es/library/554h284b(v=vs.80).aspx

```
e.Graphics.DrawCurve(Pens.Blue, points, 0.6F);
e.Graphics.DrawCurve(Pens.Green, points, 1.0F);
```



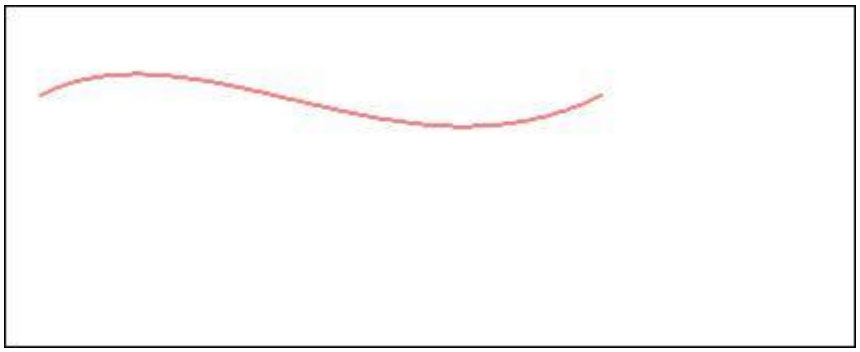
Los **Splines de Bézier**, son curvas definidas por cuatro puntos:

Punto Inicial
Punto Final
Dos Puntos de Control

Los *puntos de control* determinan la dirección de la curva en los puntos iniciales y finales. Se forman a través de una línea imaginaria que une el primer punto con el primer punto de control y el segundo punto de control con el punto final.

```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);


g.DrawBezier(Pens.Red, new Point(20, 50), new Point(100, 10), new
Point(200, 100), new Point(300, 50));
```

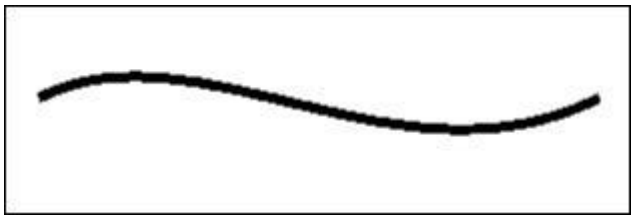


Antes de finalizar este apartado, veamos algunos detalles del objeto **Pen.**

Es un objeto que se utiliza para configurar las características que tienen las líneas al dibujar una figura. Determina, por ejemplo, el ancho de la línea, su textura o el tipo de línea. En los ejemplos anteriores se puede cambiar el argumento con la instancia del objeto **pens** por la instancia del objeto **pen**.

Por ejemplo, para crear una instancia del objeto **Pen** con líneas de color negro y un grosor de 5, escribimos el siguiente código.

```
Pen P = new Pen(Brushes.Black, 5);
```



Una vez realizados los gráficos es posible avanzar sobre otro aspecto de su diseño: el color.

2.4. Formas de Relleno

Los objetos exponen ocho métodos para hacer uso de esta funcionalidad, definiendo diferentes figuras geometrías con relleno:

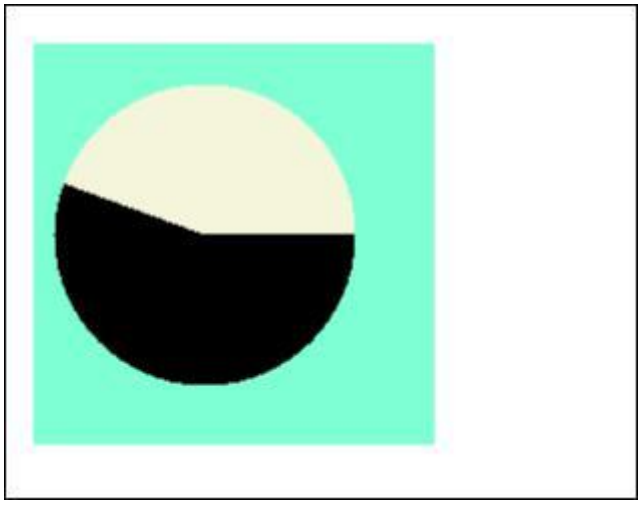
FillRectangule: Pinta Rectángulos
FillEllipse: Pinta una elipse
FillPolygon: Pinta figuras definidas por puntos
FillPie: Pinta una porción
FillClosedCurve: Especifica el modo de relleno
FillPath: Dibuja trazos complejos
FillRegion: Pinta un objeto de región

Todas aceptan un objeto **Brush** como primer argumento. El objeto **Brush**, determina la forma que se usará para rellenar la figura.

Utiliza tipos de tipos de pinceles que pueden ser sólidos, sombreados, degradados o texturas. Todos se heredan de la clase **System.Drawing.Brush**

```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);

g.FillRectangle(Brushes.Aquamarine, new Rectangle(20, 20, 200, 200));
g.FillEllipse(Brushes.Beige, new Rectangle(30, 40, 150, 150));
g.FillPie(Brushes.Black, new Rectangle(30, 40, 150, 150), 0.0F, 200.0F);
```



El método **Polygon**, se lo utiliza para rellenar formas poligonales irregulares. Acepta una matriz de objetos tipo **Point** y cada uno de los puntos define las aristas de la figura.

Puede utilizarse un argumento opcional que define el modo de relleno y puede especificar la forma en la que se interceptan las áreas. Puede elegir entre los modos de **Alternate** o **Winding**.

El modo **Alternate** es el definido por defecto, y considera la figura como cerrada. Por su parte, al usar **Winding** tiene en cuenta la dirección de los segmentos en cada intersección.

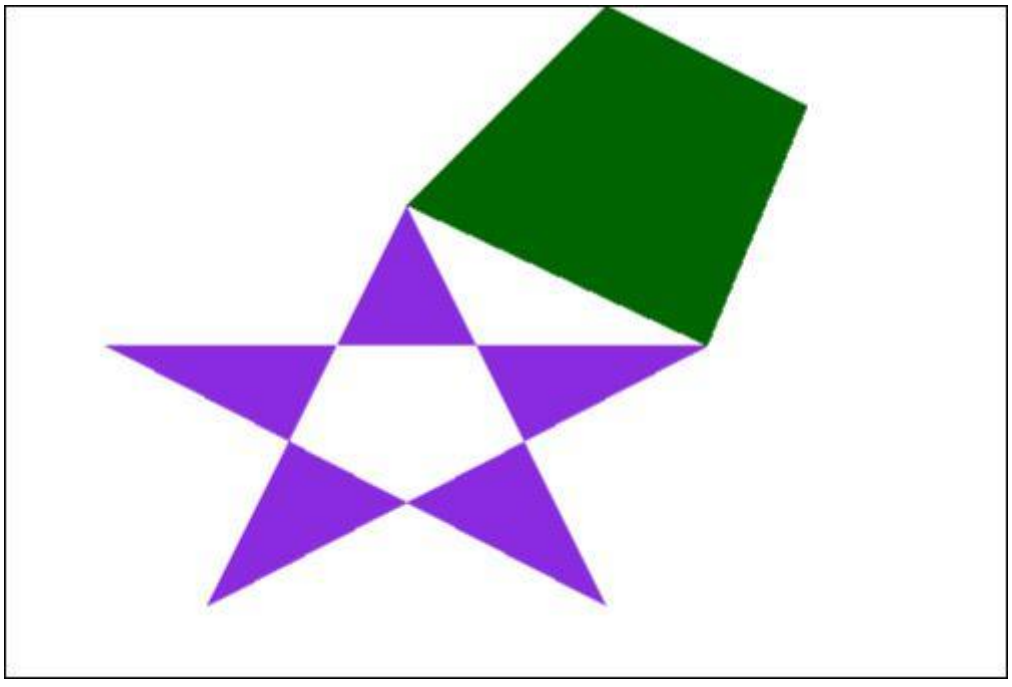
```
Graphics g = this.CreateGraphics();
g.Clear(Color.White);


'*************************************************************
'Modo alternate
'*************************************************************

 Point[]  Puntos = {new Point(200, 100), new Point(300, 300), new
Point(50, 170), new Point(350, 170), new Point(100, 300)};
g.FillPolygon(Brushes.BlueViolet, Puntos, FillMode.Alternate);


'*************************************************************
'Modo Winding
'*************************************************************

Point[] Puntos2()  = {new Point(200, 100), new Point(350, 170),
new Point(400, 50), new Point(300, 0)};
g.FillPolygon(Brushes.DarkGreen, Puntos2, FillMode.Winding);
```



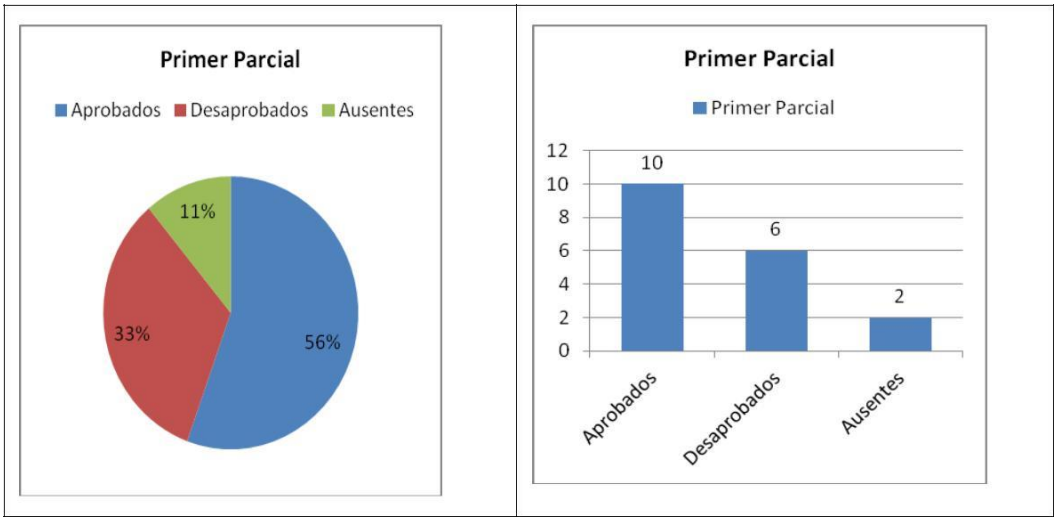
Es momento de poner en práctica lo aprendido. Creemos que usted ha seguido nuestros ejemplos con la ejercitación en máquina y es momento de proponerle la identificación de los códigos necesarios para resolver con alguna autonomía este problema.

Actividades para la facilitación de los aprendizajes

Uno de los aspectos más diferenciadores en las aplicaciones informáticas que colaboran en la toma de decisiones es la posibilidad de comparar valores gráficamente. En este ejercicio le proponemos elaborar un módulo para tal fin.

Consigna

Dado un conjunto de valores en un vector o en varios vectores, desarrolle una aplicación que grafique dichos valores y permita compararlos tal como lo muestra la siguiente figura.



Si tiene dudas o inquietudes a partir de la realización de este trabajo, por favor, compártalas con el grupo.

Espacio de comunicación e intercambio

Foro: Interfaces gráficas y pilares sobre la programación orientada a objetos

Cierre de la unidad

En esta unidad hemos visto y ejercitado las diferentes funciones que brinda .NET para trabajar graficar con facilidad. En todo sistema informatizado, el uso de estos servicios nutre de riqueza y potencialidad a los desarrollos al facilitar el uso de los medios visuales que colaboran en la toma de decisiones en el área gerencial de una empresa.

La creación de figuras elementales como rectángulos, cuadrado o círculos respeta un patrón similar que hace sencillo su aprendizaje. Dejamos en sus manos profundizar estos temas y aportar horas de práctica para superar límites y crear soluciones superlativas.

Para ampliar los contenidos de esta unidad, le sugerimos la lectura del siguiente material de consulta bibliográfica.

Finalizado este Módulo, usted se encuentra en condiciones de realizar la siguiente propuesta.

EVALUACIÓN PARCIAL

Propuesta para la Integración del Módulo II – **Utilizar técnicas avanzadas de programación**

Encontrará el documento con las consignas de la evaluación en el link correspondiente del campus virtual.

Consulte el **Cronograma de la Asignatura** para ajustar su producción a los tiempos previstos por el profesor tutor.

Recuerde que esta instancia es obligatoria y, como tal, su realización constituye un requisito para la presentación al examen final.

Realice la entrega del parcial por **Evaluaciones Parciales** para que quede registrada su actividad en el Campus Virtual y el docente pueda realizar la devolución correspondiente.

Fin de la Unidad 4 y del Módulo II – Utilizar técnicas avanzadas de programación