Основы разработки приложений с использованием Windows Forms

Урок №2

Списки, прокрутка, индикаторы

Содержание

1. Структуры Color, Size, Rectangle, Point

1.1. Структура Color

Цвета в GDI+ представлены экземплярами структуры **System.Drawing.Color** .

Первый способ создания структуры **Color** — указать значения для красного, синего и зеленого цветов, вызывая статистическую функцию **Color.FromArgb()**.

```
Color pink = Color.FromArgb(241, 105, 190);
```

Три параметра являются соответственно количествами красного, синего и зеленого цвета. Существует ряд других перегружаемых методов для этой функции, некоторые из них также позволяют определить так называемую альфа-смесь. Альфа-смешивание позволяет рисовать полупрозрачными тонами, комбинируя с цветом, который уже имеется на экране, что может создавать красивые эффекты и часто используется в играх.

Создание структуры с помощью **Color.FromArgb()** является наиболее гибкой техникой, так как она, по сути, означает, что можно определить любой цвет, который различает человеческий глаз. Однако если требуется простой, стандартный, хорошо известный цвет, такой как красный или синий, то значительно легче просто назвать требуемый цвет. В связи с этим Microsoft предоставляет также большое число статистических свойств

в **Color**, каждое из которых возвращает именованный цвет. Существует несколько сотен таких цветов. Полный список подан в документации MSDN. Он включает все простые цвета: **Red**, **Green**, **Black** и т.д, а также такие, как **DarkOrchid**, **LightCoral** и т.д.

```
Color navy = Color.Navy;
Color silver = Color.Silver;
Color sringGreen = Color.SpringGreen;
```

Структура **Color** содержит следующие методы:

Имя	Описание
`public static Color FromArgb(int argb)`	Создает структуру **Color** на основе 32-разрядного значения ARGB.
`public static Color FromArgb(int alpha, Color baseColor)`	Создает структуру **Color** из указанной структуры **Color**, но с новым определенным значением альфа. Хотя и этот метод позволяет передать 32-разрядное значение для значения альфа, оно ограничено 8 разрядами. Значение альфа для нового цвета Color допустимые от 0 до 255.
`public static Color FromArgb(int red, int green,int blue)`	Создает структуру **Color** из указанных 8-разрядных значений цветов (красный, зеленый, синий). Значение альфа неявно определено как 255 (полностью непрозрачно). Хотя и этот метод позволяет передать 32-разрядное значение для каждого компонента цвета, значение каждого из них ограничено 8 разрядами.

Имя	Описание
`public static Color FromArgb(int alpha,int red,int green,int blue)`	Создает структуру **Color** из четырех значений компонентов ARGB (альфа, красный, зеленый и синий). Хотя и этот метод позволяет передать 32-разрядное значение для каждого компонента, значение каждого из них ограничено 8 разрядами.
`public static Color FromKnownColor (KnownColor color)`	Создает структуру **Color** из указанного, предварительно определенногоцвета.
`public static Color FromName(string name)`	Создает структуру **Color** из указанного имени предопределенного цвета.
`public static Color FromSysIcv(int icv)`	Получает цвет, определенный системой. Для получения системного цвета используется целое значение.
`public float GetBrightness()`	Получает значение яркости (оттенок-насыщенность-яркость (HSB)) для данной структуры **Color**.
`public float GetHue()`	Получает значение оттенка (оттенок-насыщенность-яркость (HSB)) в градусах для данной структуры **Color**.
`public float GetSaturation()`	Получает значение насыщенности (оттенок-насыщенность-яркость (HSB)) для данной структуры **Color**.
`public int ToArgb()`	Возвращает 32-разрядное значение ARGB этой структуры **Color**.
`public KnownColor ToKnownColor()`	Возвращает значение **KnownColor** этой структуры.
`public override string ToString()`	Преобразует структуру **Color** в удобную для восприятия строку.

Операторы

Имя	Описание
`public static bool operator == (Color left, Color right)`	Проверяет эквивалентность двух указанных структур **Color**.
`public static bool operator != (Color left, Color right)`	Проверяет различие двух указанных структур **Color**.

Поля

Имя	Описание
`public static readonly Color Empty`	Представляет цвет, являющийся ссылкой **null**

Свойства

Имя	Описание
`public byte A {get;}`	Получает значение альфа-компонента этой структуры **Color**.
`public byte B {get;}`	Получает значение синего компонента этой структуры **Color**.
`public byte G {get;}`	Получает значение зеленого компонента этой структуры **Color**.
`public bool IsEmpty {get;}`	Определяет, является ли эта структура **Color** неинициализированной.
`public bool IsKnownColor{get;}`	Возвращает значение, показывающее, является ли структура **Color** предопределенным цветом. Предварительно определенные цвета, представленные элементами перечисления **KnownColor**.

Имя	Описание
`public bool IsNamedColor{get;}`	Получает значение, указывающее, является ли структура Color именованным цветом или элементом перечисления KnownColor.
`public bool IsSystemColor {get;}`	Возвращает значение, показывающее, является ли структура Color системным цветом. Системным является цвет, который используется в элементе отображения Windows. Системные цвета, представленные элементами перечисления KnownColor.
`public string Name {get;}`	Возвращает имя данного цвета Color.
`public byte R {get;}`	Получает значение красного компонента этой структуры Color.

1.2. Структура Size

Структура Size в GDI+ используется для представления размера, выраженного в пикселях. Она определяет как ширину, так и высоту.

Конструктор данной структуры перегружен:

Имя	Описание
`public Size (Point pt)`	Инициализирует новый экземпляр класса Size из указанного объекта Point.
`public Size (int width, int height)`	Инициализирует новый экземпляр класса Size из указанных размеров.

Методы

Имя	Описание
`public static Size Add(Size sz1, Size sz2)`	Прибавляет ширину и высоту одной структуры **Size** к ширине и высоте другой структуры **Size**.
`public static Size Ceiling(SizeF value)`	Преобразует указанную структуру **SizeF** в структуру **Size**, округляя значения структуры Sizedo ближайшего большего целого числа.
`public override bool Equals(Object obj)`	Проверяет, совпадают ли размеры указанного объекта **Sizec** размерами объекта **Size**.
`protected virtual void Finalize()`	Позволяет объекту **Object** попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как объект **Object** будет утилизирован в процессе сборки мусора.
`public static Size Round(SizeF value)`	Преобразует указанную структуру **SizeF** в структуру **Size**, округляя значения структуры **SizeF** до ближайших целых чисел.
`public static Size Subtract(Size sz1, Size sz2)`	Вычитает ширину и высоту одной структуры **Size** из ширины и высоты другой структуры **Size**.
`public override string ToString()`	Создает удобную для восприятия строку, представляющую размер **Size**.
`public static Size Truncate(SizeF value)`	Преобразует указанную структуру **SizeF** в структуру **Size**, округляя значения структуры **SizeF** до ближайших меньших целых чисел.

Операторы

Имя	Описание
`public static Size operator +( Size sz1, Size sz2)`	Прибавляет ширину и высоту одной структуры **Size** к ширине и высоте другой структуры **Size**.
`public static bool operator == (Size sz1, Size sz2)`	Проверяет равенство двух структур **Size**.
`public static explicit operator Point (Size size)`	Преобразует указанный размер **Size** в точку **Point**.
`public static implicit operator SizeF (Size p)`	Преобразует указанный размер **Size** в точку **SizeF**.
`public static bool operator != (Size sz1, Size sz2)`	Проверяет, различны ли две структуры **Size**.
`public static Size operator – (Size sz1, Size sz2)`	Вычитает ширину и высоту одной структуры **Size** из ширины и высоты другой структуры **Size**.

Поля

Имя	Описание
`public static readonly Size Empty`	Инициализирует новый экземпляр класса **Size**.

Свойства

Имя	Описание
`public int Height { get; set; }`	Получает или задает вертикальный компонент этого размера **Size**.
`public bool IsEmpty { get; }`	Проверяет, равны ли 0 ширина и высота размера **Size**.
`public int Width { get; set; }`	Получает или задает горизонтальный компонент размера **Size**.

1.3. Структура Point

В GDI+ структура **Point** используется для представления отдельной точки. Это точка, расположенная на двумерной плоскости, которая определяет единственный пиксель. Многим функциям, использующимся в GDI+, таким как **DrawLine()**, **Point** передается в качестве аргумента. Для структуры определены конструкторы:

```
int x = 15, y = 20;
Point p0 = new Point();
Point p1 = new Point(x);
Point p2 = new Point(x, y);
Point p3 = new Point(new Size(x, y));
```

В структуре **Point** определены следующие методы:

Имя	Описание
`public static Point Add (Point pt, Size sz)`	Добавляет заданный размер **Size** к точке **Point**.
`public static Point Ceiling (PointF value )`	Преобразует указанную структуру **PointF** в структуру **Point**, округляя значения **PointF** до ближайшего большего целого числа.
`public override bool Equals(Object obj)`	Определяет, содержит объект ли **Point** те же координаты, что и указанный объект **Object**, или нет.
`protected virtual void Finalize()`	Позволяет объекту **Object** попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как объект **Object** будет утилизирован в процессе сборки мусора.

Имя	Описание
`public void Offset(Point p)`	Смещает точку **Point** на указанную точку **Point**
`public void Offset(int dx, int dy)`	Смещает точку **Point** на указанное значение **(dx , dy)**
`public static Point Round(PointF value)`	Преобразует указанный объект **PointF** в **Point**, округляя значения Point до ближайших целых чисел.
`public static Point Subtract (Point pt, Size sz)`	Возвращает результат вычитания заданного размера **Size** из заданной точки **Point**.
`public override string ToString()`	Преобразует объект **Point** в строку, доступную для чтения.
`public static Point Truncate(PointF value)`	Преобразует указанную точку **PointF** в точку **Point** путем усечения значений объекта **Point**.

Операторы

Имя	Описание
`public static Point operator + (Point pt, Size sz)`	Смещает точку **Point** на заданное значение **Size**.
`public static bool operator == (Point left, Point right)`	Сравнивает два объекта **Point**. Результат определяет, равны значения свойств X и Y двух объектов **Point** или нет.
`public static explicit operator Size (Point p)`	Преобразует заданную структуру **Point** в структуру **Size**.
`public static implicit operator PointF (Point p)`	Преобразует заданную структуру **Point** в **PointF** структуру.

Имя	Описание
`public static bool operator != (Point left, Point right)`	Сравнивает два объекта **Point**. Результат показывает неравенство значений свойств **X** или **Y** двух объектов **Point**.
`public static Point operator – (Point pt, Size sz)`	Смещает **Point** на отрицательное значение, заданное **Size**.

Поля

Имя	Описание
`public static readonly Point Empty`	Представляет объект **Point**, у которого значения **X** и **Y** установлены равными нулю.

Свойства

Имя	Описание
`public bool IsEmpty { get; }`	Получает значение, определяющее, пуст ли класс **Point**.
`public int X { get; set; }`	Получает или задает координату X точки **Point**.
`public int Y { get; set; }`	Получает или задает координату Y точки **Point**.

1.4. Структура Rectangle

Эта структура используется в GDI+ для задания координат прямоугольников. Структура **Point** описывает верхний левый угол прямоугольника, а структура **Size** — его размеры. У **Rectangle** есть два конструктора. Одному в качестве аргументов передаются координаты

X, Y, ширина и высота. Второй конструктор принимает структуры **Point** и **Size**.

```
int x = 15, y = 20, h = 70, w = 200;
Rectangle rect0 = new Rectangle(x, y, w, h); //или
Rectangle rect1 = new Rectangle(new Point(x, y),
                  new Size(w, h));
```

Методы

Имя	Описание
`public static Rectangle Ceiling(RectangleF value)`	Преобразует указанную структуру **RectangleF** в структуру **Rectangle**, округляя значения **RectangleF** до ближайшего большего целогочисла.
`public bool Contains(Point pt)`	Определяет, содержится ли заданная точка в структуре **Rectangle**.
`public bool Contains(Rectangle rect)`	Этот метод возвращает значение **true**, если прямоугольная область, представленная параметром **rect**, полностью содержится в структуре **Rectangle**; в противном случае — значение **false**.
`public bool Contains(int x, int y)`	Определяет, содержится ли заданная точка в структуре **Rectangle**.
`public override bool Equals(Object obj)`	Проверяет, является ли **obj** структурой **Rectangle** с таким же расположением и размером, что и структура **Rectangle**.
`protected virtual void Finalize()`	Позволяет объекту **Object** попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как объект **Object** будет утилизирован в процессе сборки мусора.

Имя	Описание
`public static Rectangle FromLTRB(int left, int top, int right, int bottom)`	Создает структуру **Rectangle** с заданным положением краев.
`public void Inflate(Size size)`	Увеличивает этот **Rectangle** на указанную величину **size**.
`public void Inflate(int width, int height)`	Увеличивает этот **Rectangle** на указанную величины **width** и **height**.
`public static Rectangle Inflate(Rectangle rect, int x, int y)`	Создает и возвращает развернутую копию указанной структуры **Rectangle**. Копия увеличивается на указанную величину. Исходная структура **Rectangle** остается без изменений.
`public void Intersect(Rectangle rect)`	Заменяет данный **Rectangle** его пересечением с указанным прямоугольником **Rectangle**.
`public static Rectangle Intersect(Rectangle a, Rectangle b)`	Возвращает третью структуру **Rectangle**, представляющую собой пересечение двух других структур **Rectangle**. Если пересечение отсутствует, возвращается пустая структура **Rectangle**.
`public bool IntersectsWith (Rectangle rect)`	Определяет, пересекается ли данный прямоугольник с прямоугольником **rect**.
`public void Offset (Point pos)` `public void Offset (int x, int y)`	Этот метод изменяет положение левого верхнего угла в горизонтальном направлении, используя координату указанной точки по оси **X**, и в вертикальном направлении, используя координату этой точки по оси **Y**.

Имя	Описание
`public static Rectangle Round (RectangleF value)`	Преобразует указанный **RectangleF** в **Rectangle**, округляя значения **RectangleF** до ближайших целых чисел.
`public override string ToString()`	Преобразует атрибуты этого прямоугольника **Rectangle** в удобную для восприятия строку.
`public static Rectangle Truncate (RectangleF value)`	Преобразует указанный прямоугольник **RectangleF** в **Rectangle** путем усечения значений **RectangleF**.
`public static Rectangle Union(Rectangle a, Rectangle b)`	Получает структуру **Rectangle**, содержащую объединение двух структур прямоугольника **Rectangle**.

Операторы

Имя	Описание
`public static bool operator == (Rectangle left, Rectangle right)`	Этот оператор возвращает значение **true**, если у двух структур **Rectangle** равны свойства **X, Y, Width** и **Height**.
`public static bool operator != (Rectangle left, Rectangle right)`	Этот оператор возвращает значение **true**, если значения каких-либо из свойств **X**, **Y**, **Width** или **Height** двух структур **Rectangle** не совпадают; в противном случае — значение **false**.

Поля

Имя	Описание
`public static readonly Rectangle Empty`	Представляет структуру **Rectangle**, свойства которой не инициализированы.

Свойства

Имя	Описание
`public int Bottom { get; }`	Возвращает координату по оси **Y**, являющуюся суммой значений свойств **Y** и **Height** данной структуры **Rectangle**.
`public int Height { get;set; }`	Возвращает или задает высоту в структуре **Rectangle**.
`public bool IsEmpty { get; }`	Проверяет, все ли числовые свойства этого прямоугольника Rectangle имеют нулевые значения.
`public int Left { get; }`	Возвращает координату по оси **X** левого края структуры **Rectangle**.
`public Point Location { get; set; }`	Возвращает или задает координаты левого верхнего угла структуры **Rectangle**.
`public int Right { get; }`	Возвращает координату по оси `X`, являющуюся суммой значений свойств **X** и **Width** данной структуры **Rectangle**.
`public Size Size { get; set; }`	Возвращает или задает размер этого прямоугольника **Rectangle**.
`public int Top { get; }`	Возвращает координату по оси **Y** верхнего края структуры **Rectangle**.
`public int Width { get; set; }`	Возвращает или задает ширину структуры `Rectangle`.
`public int X { get; set; }`	Возвращает или задает координату по оси `X` левого верхнего угла структуры `Rectangle`.
`public int Y { get; set; }`	Возвращает или задает координату по оси `Y` левого верхнего угла структуры `Rectangle`.

2. Кисти

2.1. ПространствоDrawing2D

Пространство имен **System.Drawing.Drawing2D** предоставляет расширенные функциональные возможности векторной и двухмерной графики. Оно обеспечивает возможность устанавливать специальные «наконечники» для перьев, создавать кисти, которые рисуют не сплошной полосой, а текстурами, производить различные векторные манипуляции с графическими объектами.

Категория классов	Сведения
Графика и графические контуры	Классы GraphicsState и GraphicsContainer содержат информацию отчета о текущем объекте Graphics. Классы GraphicsPath представляют наборы линий и кривых. Классы GraphicsPathIterator и PathData предоставляют подробные сведения о содержимом объекта GraphicsPath.
Типы, относящиеся к матрице и преобразованию	Класс Matrix представляет матрицу для геометрических преобразований. Перечисление MatrixOrder задает порядок для матричных преобразований.
Классы Brush	Классы PathGradientBrush и HatchBrush позволяют произвести заливку форм, соответственно, с использованием градиента или шаблона штриховки.

Категория классов	Сведения
Перечисление, связанное с линиями	Перечисления LineCap и CustomLineCap позволяют задать стили завершения для линии. Перечисление LineJoin позволяет указать, как две линии объединяются в контур. Перечисление PenAlignment позволяет определить выравнивание наконечника при рисовании линии. Перечисление PenType задает шаблон, с помощью которого следует выполнить заливку линии.
Перечисления, связанные с заполнением фигур и контуров	Перечисление HatchStyle задает стили заполнения для объекта HatchStyle. Класс Blend задает шаблон смешивания для объекта LinearGradientBrush. Перечисление FillMode задает стиль заполнения для объекта GraphicsPath.

2.2. КлассBrush

Кисти (**brush**) предназначены для «закрашивания» пространства между линиями. Можно определить для кисти цвет, текстуру или изображение. Сам класс Brush является абстрактным, и создавать объекты этого класса нельзя. Класс Brush находится в пространстве имен **System.Drawing**. Производные классы TextureBrush, HatchBrush и LinearGradientBrush находятся в пространстве имен **System.Drawing.Drawing2D**. Рассмотрим более подробно наследники класса Brush.

2.3. Сплошная кисть, классSolidBrush

SolidBrush заполняет фигуру сплошным цветом. Свойство **Color** позволяет получить или задать цвет объекта SolidBrush.

2.4. Текстурная кисть, классTextureBrush

TextureBrush позволяет заполнить фигуру рисунком, хранящемся в двоичном представлении. Этот класс не может быть унаследован. При создании такой кисти также требуется задавать обрамляющий прямоугольник и режим обрамления. Обрамляющий прямоугольник определяет, какую порцию рисунка мы должны использовать при рисовании, использовать весь рисунок целиком совершенно необязательно. Приведем некоторые члены данного класса:

Имя	Описание
`public Image Image { get; }`	Получает объект Image, связанный с объектом TextureBrush.
`public Matrix Transform {get; set;}`	Получает или задает копию объекта Matrix, определяющую локальное геометрическое преобразование для изображения, связанного с объектом TextureBrush.
`public WrapMode WrapMode {get; set;}`	Получает или задает перечисление WrapMode, задающее режим переноса для объекта TextureBrush.

Методы

Имя	Описание
`public void MultiplyTransform (Matrix matrix)` `public void MultiplyTransform (Matrix matrix, MatrixOrder order)`	Умножает объект **Matrix**, представляющий локальное геометрическое преобразование объекта **TextureBrush**, на указанный объект **Matrix** с помощью добавления указанного объекта **Matrixв** начало.
`public void ResetTransform()`	Возвращает свойству **Transform** объекта **TextureBrush** единичное значение.
`public void RotateTransform (float angle)`	Поворачивает локальное геометрическое преобразование объекта **TextureBrush** на заданную величину. Этот метод добавляет поворот перед преобразованием.
`public void RotateTransform (float angle, MatrixOrder order)`	Поворачивает локальное геометрическое преобразование объекта **TextureBrush** назаданную величину в указанномпорядке.
`public void ScaleTransform (float sx, float sy)` `public void ScaleTransform (float sx, float sy, MatrixOrder order)`	Изменяет масштаб локального геометрического преобразования объекта **TextureBrush** на заданные значения. Этот метод вставляет изменение масштаба перед преобразованием. **sx** — Величина изменения масштаба по оси **X**. **sy** — Величина изменения масштаба по оси **Y**. **order** — Перечисление **MatrixOrder**, задающее порядок использования матрицы изменения масштаба (в начале или в конце).

Имя	Описание
`public void TranslateTransform (float dx, float dy)` `public void TranslateTransform (float dx, float dy, MatrixOrder order)`	Выполняет перенос локального геометрического преобразования объекта `TextureBrush` на заданные значения в указанном порядке. **dx** — Величина переноса преобразования по оси **X**. **dy** — Величина переноса преобразования по оси **Y**. **order** — Порядок применения преобразования (в начале или в конце).

2.5. Кисть с насечками, класс HatchBrush

Более сложное «закрашивание» можно произвести при помощи производного от **Brush** класса **HatchBrush**. Этот тип позволяет закрасить внутреннюю областью объекта при помощи большого количества штриховки, определенных в перечислении **HatchStyle** (имеется 54 уникальных стиля штриховки). Основной цвет задает цвет линий; цвет фона определяет цвет интервалов между линиями. Этот класс не может быть унаследован.

Свойства (выборочно)

Имя	Описание
`public Color BackgroundColor { get; }`	Получает цвет интервалов между линиями штриховки, нарисованными данным объектом `HatchBrush`.
`public Color ForegroundColor { get; }`	Получает цвет линий штриховки, нарисованных данным объектом `HatchBrush`.
`public HatchStyle HatchStyle { get; }`	Получает стиль штриховки для данного объекта `HatchBrush`.

2.6. Градиентная кисть, класс LinearGradientBrush

LinearGradientBrush содержит кисть, которая позволяет рисовать плавный переход от одного цвета к другому, причем первый цвет переходит во второй под определенным углом. Углы при этом задаются в градусах. Угол, равный 0°, означает, что переход от одного цвета к другому осуществляется слева направо. Угол, равный 90° , означает, что переход от одного цвета к другому осуществляется сверху вниз. Этот класс не может быть унаследован. Приведем некоторые свойства **LinearGradientBrush**:

Свойства

Имя	Описание
`public Blend Blend { get; set; }`	Получает или задает объект **Blend**, определяющий позиции и коэффициенты, задающие настраиваемый спад для градиента.
`public bool GammaCorrection { get; set; }`	Получает или задает значение, указывающее, включена ли гамма-коррекция для этого объекта **LinearGradientBrush**.
`public ColorBlend InterpolationColors { get; set; }`	Получает или задает перечисление **WrapMode**, задающее режим переноса для объекта **TextureBrush**.
`public Color[] LinearColors { get; set; }`	Получает или задает начальный и конечный цвета градиента.
`public RectangleF Rectangle { get; }`	Получает прямоугольную область, которая определяет начальную и конечную точки градиента.

2.7. Кисть с использованием траектории, классPathGradientBrush

`PathGradientBrush` позволяет создавать сложный эффект затенения, при котором используется изменение цвета от середины рисуемого пути к его краям.

Свойства (выборочно)

Имя	Описание
`public Color CenterColor { get; set; }`	Получает или задает цвет в центре градиента контура.
`public PointF CenterPoint { get; set; }`	Получает или задает центральную точку градиента контура.
`public PointF FocusScales {get; set; }`	Получает или задает точку фокуса для гради ентного перехода.
`public ColorBlend InterpolationColors { get; set; }`	Получает или задает объект **ColorBlend**, определяющий много-цветный линейный градиент.
`public Color [] SurroundColors { get; set; }`	Получает или задает массив цве-тов, соответствующих точкам кон-тура, заполняемого объектом **Path-GradientBrush**.

3. Перья, класс Pen

Обычное применение объектов **Pen** (перьев) заключается в рисовании линий. Как правило, объект **Pen** используется не сам по себе: он передается в качестве параметра многочисленным методам вывода, определенным в классе **Graphics**.

В этом классе предусмотрены несколько перегруженных конструкторов, при помощи которых можно задать исходный цвет и толщину пера. Большая часть возможностей **Pen** определяется свойствами этого класса.

Свойства (выборочно)

Имя	Описание
`public PenAlignment Alignment {get; set;}`	Это свойство определяет, каким образом объект **Pen** рисует замкнутые кривые и многоугольники. Перечислением **PenAlignment** задаются пять значений; однако только два значения — **Center** и **Inset** — изменяют внешний вид рисуемой линии. Значение **Center** устанавливается по умолчанию для этого свойства, и им задается центрирование ширины пера по контуру кривой или многоугольника. Значение **Inset** этого свойства соответствует размещению пера по всей ширине внутри границы кривой или многоугольника. Три другие значения, **Right**, **Left** и **Outset** задают центрированное перо.

Имя	Описание
	Перо **Pen**, для которого выравнивание установлено равным **Inset**, выдает недостоверные результаты, иногда рисуя в позиции вставки, а иногда — в центрированном положении. К тому же, вложенное перо невозможно использовать для рисования составных линий, и оно не позволяет рисовать пунктирные линии с оконечными элементами **Triangle**.
`public Brush Brush { get; set; }`	Получает или задает объект **Brush**, определяющий атрибуты объекта **Pen**.
`public float[] CompoundArray { get; set; }`	Получает или задает массив значений, определяющий составное перо. Составное перо рисует составную линию, состоящую из параллельных линий и разделяющих их промежутков.
`public CustomLineCap CustomStartCap { get; set; }`	Получает или задает настраиваемое начало линий, нарисованных при помощи объекта **Pen**.
`public DashStyle DashStyle { get; set; }`	Получает или задает стиль, используемый для пунктирных линий, нарисованных при помощи объекта **Pen**.
`public PenType PenType { get;}`	Получает или задает стиль линий, нарисованных с помощью объекта **Pen**.

Перечисление DashStyle

Имя члена	Описание
`Solid`	Задает сплошную линию.
`Dash`	Задает линию, состоящую из штрихов.
`Dot`	Задает линию, состоящую из точек.

Имя члена	Описание
DashDot	Задает штрих-пунктирную линию.
DashDotDot	Задает линию, состоящую из повторяющегося шаблона “штрих-две точки”.
Custom	Задает пользовательский тип пунктирных линий.

Кроме класса **Pen** в GDI+ также можно использовать коллекцию заранее определенных перьев (коллекция **Pens**). При помощи статистических свойств коллекции **Pens** можно мгновенно получить уже готовое перо, без необходимости создавать его вручную. Однако все типы Pen, которые создаются при помощи коллекции **Pens**, имеют одинаковую ширину равную 1. Изменить ее нельзя.

Для работы с «наконечниками» перьев служит перечисление **LineCap**.

Значения перечисления **LineCap**.

Значение	Описание
ArrowAnchor	Линии оканчиваются стрелками
DiamondAnchor	Лини оканчиваются ромбами
Flat	Стандартное прямоугольное завершение линий
Round	Лини на концах скруглены
RoundAnchor	На концах линий — «шары»
Square	На концах линий — квадраты в толщину линии
SquareAnchor	На концах линий — квадраты большего размера, чем толщина линий.
Triangle	Треугольное завершение линий.

Желательно вызывать метод **Dispose()** для создаваемых объектов **Brush** и **Pen**, либо использовать для работы с ними конструкцию **using**, в противном случае приложение может исчерпать ресурсы системы.

4. Примеры использования кистей и перьев

Рассмотрим пример программы с использованием кистей и перьев разных видов. Создадим статический класс **Cub**, реализовывающий прорисовку создание графического куба и разные методы заливки граней. В методе **BrushesExampleMethod(Graphics g)** показана работа с кистями. Создадим структуру **Color** с помощью метода **Color.FromArgb()**:

```
Color pink = Color.FromArgb(241, 105, 190);

//создаем сплошную кисть цвета pink
SolidBrush sldBrush = new SolidBrush(pink);

//закрасим кистью sldBrush прямоугольник, у которого
//верхний левый угол имеет координаты (300;150),
//а высоту и широту 70
g.FillRectangle(sldBrush, 300, 150, 70, 70);
```

Рассмотрим применение штриховки **hBrush**. В качестве параметров конструктору передаем стиль штриховки, цвет линий и цвет промежутков между линиями:

```
HatchBrush hBrush =
                    new HatchBrush(HatchStyle.
                    NarrowVertical,
                    Color.Pink, Color.Blue);
g.FillRectangle(hBrush, 370, 150, 70, 70);
```

Создадим градиентную кисть **lgBrush**. Структура **Rectangle** представляет границы линейного градиента:

```
LinearGradientBrush lgBrush = new LinearGradientBrush(new
        Rectangle(0, 0, 20, 20), Color.Violet, Color.
        LightSteelBlue, LinearGradientMode.Vertical);
g.FillRectangle(lgBrush, 300, 220, 70, 70);
```

Для заливки следующего прямоугольника будем использовать встроенную кисть из коллекции **Brushes**:

```
g.FillRectangle(Brushes.Indigo, 370, 220, 70, 70);
```

Воспользуемся текстурной кистью. Рисунок для кисти загрузим из файла:

```
TextureBrush tBrush = new
TextureBrush(Image.FromFile(@"Images\charp.jpg"));
g.FillRectangle(tBrush, 370, 290, 70, 70);
```

В методе **DrawLeftAndUpperBound** класса **Cub** рисуем левую и верхнюю грань куба:

```
public static void DrawLeftAndUpperBound(Graphics g)

{
Point[] p = { new Point(240, 110), new Point(440, 110),
        new Point(510, 150), new Point(300, 150) };
TextureBrush tBrush = new
        TextureBrush(Image. FromFile(@"Images\0073.jpg"));
g.FillPolygon(tBrush, p);

Point[] p1 = { new Point(240, 110), new Point(300, 150),
        new Point(300, 360), new Point(240, 310) };
```

```
HatchBrush hBrush = new HatchBrush(HatchStyle.
        DashedDownwardDiagonal,
        Color.Violet, Color.White);
g.FillPolygon(hBrush, p1);
}
```

Работа с перьями демонстрируется в методе **PensExampleMethod** класса **Cub**:

```
public static void PensExampleMethod(Graphics g)
{
//рисуем вертикальные линии сплошным пером
    Pen p = new Pen(Color.White, 1);
    for (int i = 0; i < 4; i++)
    {
        g.DrawLine(p,300+70*i,150,300+70*i,360);
    }
//рисуем горизонтальные линии пунктирным пером
    p = new Pen(Color.White, 1);
    p.DashStyle = DashStyle.DashDot;
    for (int i = 0; i < 4; i++)
    {
         g.DrawLine(p, 300, 150 + 70 * i, 510,
                    150 + 70 * i);
    }
//рисуем косые линии пером, на концах которого ромбы
    p = new Pen(Color.White, 1);
    p.StartCap = LineCap.DiamondAnchor;
    p.EndCap = LineCap.DiamondAnchor;
    for (int i = 0; i < 3; i++)
    {
        g.DrawLine(p, 240+70*i, 110 , 300+70*i, 150);
    }
}
```

5. Изображения. Типы изображений

Существует два типа графики — растровая и векторная. Основное отличие — в принципе хранения изображения.

Растровый рисунок можно сравнить с мозаикой, когда изображение разбито на небольшие одноцветные части. Эти части называют пикселями (**PICture ELement** — элемент рисунка). Чем выше разрешение изображения (число пикселей на единицу длины), тем оно качественнее. Но изображение с высоким разрешением занимает много дисковой памяти, а для его обработки требуется много оперативной памяти. Кроме того, растровые изображения трудно масштабировать. При уменьшении — несколько соседних пикселей преобразуются в один, поэтому теряется разборчивость мелких деталей. При увеличении — увеличивается размер каждого пикселя, поэтому появляется эффект" лоскутного одеяла».

Векторная графика описывает изображение с помощью графических примитивов, которые рассчитываются по конкретным математическим формулам. Сложное изображение можно разложить на множество простых объектов. Любой такой простой объект состоит из контура и заливки.

Основное преимущество векторной графики состоит в том, что при изменении масштаба изображение не теряет своего качества. Отсюда следует и другой вывод — при изменении размеров изображения не изменяется размер

файла. Ведь формулы, описывающие изображение, остаются те же, меняется только коэффициент пропорциональности.

Однако если делать много очень сложных геометрических фигур, то размер векторного файла может быть гораздо больше, чем его растровый аналог из-за сложности формул, описывающих такое изображение.

Следовательно, векторную графику следует применять для изображений, не имеющих большого числа цветовых фонов, полутонов и оттенков. Например, создания пиктограмм, логотипов, иллюстраций, рекламных модулей и т.д.

5.1. КлассImage

Тип **System.Drawing.Imaging** определено множество типов для проведения сложных преобразований изображений.

Класс **Image** является абстрактным, и создавать объекты этого класса нельзя. Он предоставляет функциональные возможности для производных классов **Bitmap** и **Metafile**. Обычно объявленные переменные **Image** присваиваются объектам класса **Bitmap**.

Приведем наиболее важные члены класса **Image**, многие из них являются статическими, а некоторые — абстрактные.

Методы

Имя	Описание
`public static Image FromFile(string filename)`	Создает объект **Image** из указанного файла, используя внедренную информацию управления цветом из файла.

Имя	Описание
`public static Image FromFile(string filename, bool useEmbedded ColorManagement)`	**useEmbeddedColorManagement** — устанавливается в **true** для использования информации управления цветом, внедренной в файл с изображением; иначе устанавливается.
`public RectangleF GetBounds (ref GraphicsUnit pageUnit)`	Получает границы изображения в заданных единицах измерения.
`public void Save (string filename)`	Перегружен. Сохраняет объект **Image** в указанный файл или поток.
`public int SelectActiveFrame (FrameDimension dimension, int frameIndex)`	Выделяет кадр, заданный размером и индексом.

Свойства

Имя	Описание
`public int Height {get; }`	Получает высоту объекта **Image** в точках.
`public float HorizontalResolution {get; }`	Получает горизонтальное разрешение объекта **Image** в точках на дюйм.
`public ColorPalette Palette {get; set; }`	Возвращает или задает палитру цветов, используемую для объекта **Image**.
`public ImageFormat RawFormat { get; }`	Получает формат файла объекта **Image**.
`public Object Tag { get; set; }`	Возвращает или задает объект, предоставляющий дополнительные данные об изображении.

Имя	Описание
`public float VerticalResolution { get; }`	Получает вертикальное разрешение объекта `Image` в точках на дюйм.
`public int Width { get; }`	Получает ширину объекта `Image` в точках.

5.2. Класс Bitmap

Инкапсулирует точечный рисунок GDI+, состоящий из данных точек графического изображения и атрибутов рисунка. Объект **Bitmap** используется для работы с растровыми изображениями.

Данный класс имеет 12 конструкторов. Следующие конструкторы загружают объект **Bitmap** из файла, потока или ресурса:

Имя	Описание
`public Bitmap (String filename)`	Инициализирует новый экземпляр класса **Bitmap** из указанного файла **filename**.
`public Bitmap (Stream stream)`	Инициализирует новый экземпляр класса **Bitmap** из указанного потока данных **stream**.
`public Bitmap (Image original)`	Инициализирует новый экземпляр класса **Bitmap** из указанного существующего изображения **original**.
`public Bitmap (Type type, String resource)`	Инициализирует новый экземпляр класса **Bitmap** из указанного ресурса **resource**.

Для вывода изображения необходимо иметь подходящий экземпляр **Graphics**. Достаточно вызвать метод **Graphics**.

DrawImage(). Доступно сравнительно немного перегрузок этого метода, но они обеспечивают гибкость за счет указания информации относительно местоположения и размера отображаемого образа.

Уничтожить изображение сразу после того, как отпадает в нем необходимость важно потому, что обычно графические изображения используют большие объемы памяти. После того, как вызван **Bitmap.Dispose**(), экземпляр **Bitmap** более не ссылается ни на какое реальное изображение, а потому не может его отображать.

Один из вариантов метода **DrawImage**() принимает объекты **Bitmap** и **Rectangle**. Прямоугольник задает область, в которой должно быть нарисовано изображение. Если размер прямоугольника назначения отличается от размеров исходного изображения, изображение масштабируется, чтобы соответствовать прямоугольнику назначения.

Некоторые варианты метода **DrawImage**() получают в качестве параметров не только конечный, но и исходный прямоугольник. Исходный прямоугольник задает часть исходного изображения, которая должна быть нарисована. Прямоугольник назначения задает прямоугольник, в котором должна быть нарисована эта часть изображения. Если размер прямоугольника назначения отличается от размера исходного прямоугольника, изображение масштабируется, чтобы соответствовать размеру прямоугольника назначения.

Версии **DrawImage**(), ориентированные на прямоугольник, могут не только масштабировать изображение, но и кое-что другое. Если указать отрицательную ширину, картинка будет повернута по вертикальной оси, и вы

получите ее зеркальную копию. При отрицательной высоте метод поворачивает ее по горизонтальной оси и отображает вверх ногами. В любом случае верхний левый угол исходного, неперевернутого изображения позиционируется согласно координатам **Point** и **PointF** прямоугольника, указанными в **DrawImage**().

Методы (выборочно)

Имя	Описание
`public static Bitmap FromHicon(IntPtr hicon)`	Создает изображение **Bitmap** для значка из дескриптора Windows.
`public static Bitmap FromResource (IntPtr hinstance, string bitmapName)`	Создает изображение **Bitmap** из указанного ресурса Windows.
`public void SetPixel (int x, int y, Color color)`	Задает цвет указанной точки в этом изображении **Bitmap**.
`public void SetResolution(float xDpi, float yDpi)`	Устанавливает разрешение для этого изображения **Bitmap**.
`public void UnlockBits (BitmapData bitmapdata)`	Разблокирует это изображение **Bitmap** из системной памяти.

Сохранение изображений осуществляется с помощью метода **Save**, который имеет несколько перегрузок:

Имя	Описание
`public void Save(string filename)`	Сохраняет объект **Bitmap** в указанный файл или поток **filename**.
`public void Save(Stream stream, ImageFormat format)`	Сохраняет данное изображение в указанный поток **stream** в указанном формате **format**.

Имя	Описание
`public void Save(string filename, ImageFormat format)`	Сохраняет объект **Image** в указанный файл в указанном формате.
`public void Save(Stream stream, ImageCodecInfo encoder, EncoderParameters encoderParams)`	Сохраняет данное изображение в указанный поток с заданным кодировщиком и определенными параметрами кодировщика изображения.
`public void Save(string filename, ImageCodecInfo encoder, EncoderParameters encoderParams)`	Сохраняет объект **Image** в указанный файл с заданным кодировщиком и определенными параметрами кодировщика изображения.

Метод **Save** не работает с объектами **Image**, загруженными из метафайла или хранящимися в битовой карте в памяти. Кроме того, нельзя сохранить изображение в формате метафайла или битвой карты в памяти.

Рассмотрим использования изображений на следующем примере.

Объявим переменную **picture** класса **Image** и загрузим изображение из имеющегося на диске графического файла:

```
picture = Image.FromFile(@"Images\metrobits.jpg");
```

Зададим область, в которой будет выводиться наше изображение:

```
//Параллелограмм в котором будет выведено изображение
pictureBounds = new Point[3];
pictureBounds[0] = new Point(0, 0);
pictureBounds[1] = new Point(picture.Width, 0);
pictureBounds[2] = new Point(picture.Width/3,
                             picture.Height);
```

Проведем преобразования и выведем рисунок на форму:

```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    Graphics graphics = e.Graphics;
    graphics.ScaleTransform(1.0f, 1.0f);
    graphics.TranslateTransform(this.
             AutoScrollPosition.X,
             this.AutoScrollPosition.Y);
    graphics.DrawImage(picture, pictureBounds);
}
```

5.3. КлассMetafile

Метафайл состоит из последовательностей двоичных записей, соответствующих вызовам графических функций, рисующих прямые и кривые линии, закрашенные фигуры и текст. Кроме того, метафайлы могут содержать встроенные растровые изображения. Метафайлы могут храниться на жестком диске или целиком располагаться в памяти.

Поскольку метафайл описывает рисунок с помощью команд рисования графических объектов, такой рисунок можно масштабировать, не теряя разрешения, в отличие от растровых изображениях. Если увеличить размер растрового изображения, его разрешение останется прежним, так как пиксели изображения просто копируются по вертикали и горизонтали. Любое сглаживание, применяемое при отображении масштабированных растровых изображений, убирает неровности, но при этом делает картинку размытой.

Метафайл можно преобразовать в растровое изображение, но при этом часть данных теряется: графические

объекты, из которых состоял рисунок, объединяются в одно изображение и не могут изменяться по отдельности. Преобразование растровых изображений в метафайлы представляет собой гораздо более сложную задачу и обычно применяется только для очень простых изображений — ведь для определения границ и контуров нужна большая вычислительная мощность.

Сегодня метафайлы чаще всего применяются для передачи рисунков между программами через буфер обмена и для иллюстративных вставок. Поскольку метафайлы описывают рисунок в виде вызовов графических функций, они занимают меньше места и менее аппаратно зависимы, чем растровые изображения.

В C# метафайлы представлены классом **Metafile** из пространства имен **System.Drawing.Imaging**. Для загрузки метафайла с диска можно использовать тот же статистический метод **FromFile** класса **Image**, что и для растрового изображения. Этот метод имеет несколько перегрузок.

Большую часть класса **Metafile** составляют 39 конструкторов. Создать объект **Metafile**, сославшись на существующий метафайл по имени файла или объекта типа **Stream**, можно, используя конструкторы:

Имя	Описание
`public Metafile (String filename)`	Инициализирует новый экземпляр класса **Metafile** из указанного файла **filename**.
`public Metafile (Stream stream)`	Инициализирует новый экземпляр класса **Metafile** из указанного потока данных **stream**.

Эти два конструктора во многом эквивалентны соответствующим статическим методам **FromFile** класса **Image** за исключением того, что они явно возвращают объект типа **Metafile**.

Так как класс **Metafile** наследуются от **Image**, для отображения метафайла используются такие же методы **Graphics**.**DrawImage**().

Разумеется, по отношению к метафайлу можно применять любые действия, поддерживаемые классом **Image**. Однако, если вы загрузили метафайл из файла или потока, вам не удастся задействовать статистический метод **DrawImage** класса **Graphics**, чтобы получить объект **Graphics** для изменения метафайла. Этот метод зарезервирован для новых метафайлов, создаваемых вашей программой.

Для сохранения метафайлов используется метод **Save**, наследуемый от **Image**.

При работе с метафайлами особенно полезны:

Имя	Описание
`public int Height { get; }`	Получает высоту объекта `Image` в точках.
`public int Width { get; }`	Получает ширину объекта Imagев точках.
`public float HorizontalResolution { get; }`	Получает горизонтальное разрешение объекта `Image` в точках на дюйм.
`public float VerticalResolution { get; }`	Получает вертикальное разрешение объекта `Image` в точках на дюйм.
`public Size Size { get; }`	Получает ширину и высоту данного изображения в точках.

Класс **Metafile** не содержит дополнительных открытых свойств, кроме тех, что он наследует от класса **Image**. Однако в самом метафайле существует заголовок, содержащий дополнительные сведения о файле. Этот заголовок инкапсулируется в классе **MetafileHeader**. Получить экземпляр этого класса позволяет нестатический метод **GetMetafileHeader()** класса **Metafile**.

Для примера работы с векторными изображениями рекомендуется разобрать листинг следующего примера:

```
public partial class Form1 : Form
{
private Metafile wmfImage;
public Form1()
{
     InitializeComponent();
     LoadWMFFile(@"Images\Graphic.wmf");
}

//Открытие файла
private void tsmiOpenFile_Click(object sender,
               System.EventArgs e)
{
     OpenFileDialog ofg = new OpenFileDialog()
     {
          CheckFileExists = true,
          CheckPathExists = true,
          ValidateNames = true,
          Title = "Open WMF File",
          Filter = "WMF files (*.wmf)|*.wmf"
     };
if (ofg.ShowDialog() == DialogResult.OK)
     {
          LoadWMFFile(ofg.FileName);
     }
```

```
}

//Перегрузка перерисовки формы
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);

    //Определение минимальной составляющей
    //размера формы
    double minFormBound = Math.Min(this.
        ClientSize.Width, this.ClientSize.Height);

    //Определение максимальной составляющей
    //размера рисунка
    double maxPictureDimension = Math.
        Max(wmfImage.Width, wmfImage.Height);

    //Определение коефициента изменения размеров
    //рисунка
    double k = maxPictureDimension / minFormBound;

    int w = (int)(wmfImage.Width / k);
    int h = (int)(wmfImage.Height / k);

    //Создание координат вывода рисунка
    Point[] imageBounds = new Point[3];
    imageBounds[0] = new Point(50, 50);
    imageBounds[1] = new Point(w, 50);
    imageBounds[2] = new Point(50, h);
    //Вывод рисунка на форму
    e.Graphics.DrawImage(wmfImage, imageBounds);
}

//Загрузка рисунка из файла
private void LoadWMFFile(String name)
{
```

```
    //Создание рисунка из файла
    this.wmfImage = new Metafile(name);
    //Вызов метода перерисовки формы
    this.Invalidate();
}

//Создание нового рисунка, смайла
private void CreateWMFFile()
{
    Graphics formGraphics = this.CreateGraphics();
    IntPtr ipHdc = formGraphics.GetHdc();

    wmfImage.Dispose();
    wmfImage = new Metafile(@"Images\New.wmf", ipHdc);
    formGraphics.ReleaseHdc(ipHdc);
    formGraphics.Dispose();

    Graphics graphics = Graphics.FromImage(wmfImage);

    graphics.FillEllipse(Brushes.Yellow, 100, 100,
                         200, 200);
    graphics.FillEllipse(Brushes.Black, 150, 150,
                         25, 25);
    graphics.FillEllipse(Brushes.Black, 225, 150,
                         25, 25);
    graphics.DrawArc(new Pen(Color.SlateGray, 10),
                         195, 180, 10, 100, -30, -120);
    graphics.DrawArc(new Pen(Color.Red, 10), 150,
                         170, 100, 100, 30, 120);
    graphics.Dispose();
    this.Invalidate();
}
private void tsmiNew_Click(object sender, EventArgs e)
{
    CreateWMFFile();
}
```

```
private void Form1_Resize(object sender, EventArgs e)
{
    this.Invalidate();
}
}
```

6. Шрифты

Характеристики шрифтов

Шрифт — это система визуального отображения информации при помощи условных символов. В более узком значении это комплект литер, цифр и специальных знаков определенного рисунка.

Есть несколько характеристик шрифтов:

- *Кегль шрифта* (размер шрифта — высота в типографских пунктах прямоугольника, в который может быть вписан любой знак алфавита данного размера с учетом верхнего и нижнего просвета): текстовые (до 12 пунктов), титульные (более 12 пунктов).
- *Гарнитура шрифта* (комплект шрифтов одинакового рисунка, но различного начертания и размера). Имеют условные названия: литературная, обыкновенная, плакатная и др.
- *Начертание шрифтов* (насыщенность и толщина штрихов, высота знаков и характер заполнения): светлые, полужирные и жирные.
- *Наклон основных шрифтов* (отклонение основных штрихов от вертикального положения): прямые, курсивные.
- *Размер шрифта* (в нормальных шрифтах отношение ширины очка к высоте составляет приблизительно 3:4, в узких — 1:2, в широких — 1:1): сверхузкие, узкие, нормальные, широкие и

сверхширокие — характер заполнения штрихов: шрифт нормальный, контурный, выворотный, оттененный, штрихованный и др.

«Компьютерный» шрифт — это файл или группа файлов, обеспечивающий вывод текста со стилевыми особенностями шрифта. Обычно система файлов, составляющая шрифт, состоит из основного файла, содержащего описание символов и вспомогательных информационных и метрических файлов, используемых прикладными программами. Пользователи имеют возможность использовать как растровые, так и векторные шрифты. Файлы растровых шрифтов содержат описания букв в виде матриц растра последовательность печатаемых точек. Каждому кеглю какого-либо начертания растрового шрифта соответствует файл на диске, используемый программой при печати, поэтому для растровых шрифтов часто используется термин шрифторазмер.

С выходом MS Windows 3.1 (1992 г.) приложения Windows стали использовать шрифты по-новому. Раньше большинство шрифтов, пригодных для отображения на экране монитора в Windows, были растровыми (точечными). Имелись также штриховые шрифты (их также называют плоттерными или векторными), которые определялись как полилинии (polylines), но они имели малопривлекательный вид и использовались редко. В Windows 3.1 была представлена технология TrueType— это технология контурных шрифтов, созданная Apple и Microsoft и поддерживаемая большинством производителей шрифтов. Контурные шрифты можно плавно

масштабировать, они содержат встроенную разметку (hints), которая не допускает искажений; контуры масштабируются в соответствии с определенным размером пикселя и координатной сеткой.

В 1997 г. Adobe и Microsoft анонсировали формат шрифтов OpenType. Этот формат сочетает TrueType и формат контурных шрифтов Type 1, используемый и PostScript — языке описания страниц фирмы Adobe.

Программы с Windows Forms имеют прямой доступ только к шрифтам TrueType и OpenType.

Впервые примененная в Windows, технология TrueType была представлена файлами

TrueType (с расширением .ttf). Это были шрифты:

- CourierNew;
- Times NewRoman;
- Arial;
- Symbol,

- *Courier* — это семейство моноширинных шрифтов, стилизующих печать на пишущей машинке. В наши дни этот шрифт обычно используется только в текстовых окнах, листингах программ и сообщениях об ошибках,
- *Times New Roman* — это разновидность гарнитуры Times (переименованная по юридическим соображениям). Изначально она была создана для газеты «Times of London». Считается, что текст, набранный этой гарнитурой, удобен для чтения.
- *Arial* — это разновидность гарнитуры Helvetica, популярного рубленого шрифта (sans serif). Серифы

(serifs) — это небольшие засечки на концах линий. Рубленый шрифт не содержит таких засечек. (Шрифты с засечками иногда называют прямыми или латинскими (roman).) Шрифт Symbol включает не буквы, а часто используемые символы.

6.2. Класс Font

В пространстве имен **System.Drawing** определены два важных класса для работы со шрифтами:

- **FontFamily** определен как строка, например «Times New Roman »;
- **Font**— это комбинация из названия шрифта (объект **FontFamily** или символьная строка, определяющая гарнитуру), атрибутов (таких как курсив или полужирный) и кегля;

Рассмотрим класс **Font**. Он включает в себя три основные характеристики шрифта, а именно: семейство, размер и стиль. Есть три категории конструкторов **Font**, основанные на:

- существующем объекте **Font**;
- символьной строке, определяющейгарнитуру:
- объекте **FontFamily**.

Простейший конструктор **Font** создает новый шрифт на базе существующего. Новый шрифт отличается лишь начертанием:

- **Font** (**Font font**, **FontStyle fs**).
- **FontStyle** — это перечисление, состоящие из серии однобитных флагов.

Перечисление **FontStyle**:

Член	Значение
Regular	0
Bold	1
Italic	2
Underline	4
Strikeout	8

Следующий набор конструкторов класса **Font** очень удобен: вы определяете шрифт, выбирая гарнитуру, кегль и, если требуется, начертание:

```
Font(string strFamily, float fSizelnPoints)
Font(string strFamily, float fSizeInPoints, FontStyle fs)
```

Свойство **Size** определяет размер данного шрифта. Однако в .Net Framework это свойство не является обязательно размером шрифта, выраженным в пунктах. Существует возможность изменить свойство **GraphicsUnit** (единица измерения графических объектов) посредством свойства **Unit**, которое определяет единицу измерения шрифтов. С помощью перечислимого типа **GraphicsUnit** размер шрифта может задаваться с помощью следующих единиц измерения:

- пункты;
- особый шрифт (1/75дюйма);
- документ (1/300дюйма);
- дюйм;
- миллиметр;
- пиксель.

Также класс **Font** содержит следующие члены (методы выборочно):

Имя	Описание
`public static Font FromHdc(IntPtr hdc)`	Создает шрифт **Font** из указанного дескриптора Windows для контекста устройства.
`public static Font FromHfont (IntPtr hfont)`	Создает шрифт **Font** из указанного дескриптора Windows.
`public float GetHeight()`	Возвращает значение междустрочного интервала данного шрифта в точках.

Свойства (выборочно)

Имя	Описание
`public bool Bold {get; }`	Возвращает значение, определяющее, является ли этот шрифт **Font** полужирным.
`public FontFamily FontFamily { get; }`	Возвращает семейство шрифтов **FontFamily**, связанный с данным шрифтом **Font**.
`public int Height { get; }`	Возвращает значение междустрочного интервала данного шрифта.
`public bool Italic { get; }`	Возвращает значение, определяющее, является ли этот шрифт **Font** курсивом.
`public string Name { get; }`	Возвращает имя начертания этого шрифта **Font**.
`public bool Strikeout { get; }`	Возвращает значение, указывающее, задает ли данный шрифт **Font** горизонтальную линию через шрифт.
`public bool Underline { get; }`	Возвращает значение, показывающее, является ли этот шрифт **Font** подчеркнутым.

6.3. Примеры использования шрифтов

Рассмотрим пример использования шрифтов.

Выведем на форму строку шрифтом «Comic Sans MS». Для этого создадим объект **font** класса **Font** и передадим его в метод **DrawStr**, который непосредственно занимается выводом строки на форму:

```
private void Form1_Shown(object sender, EventArgs e)
{
    Font font = new Font("Comic Sans MS", 16,
            FontStyle.Italic);
    DrawStr(font, "Строка, которая будет выведена
            на форму");
}

private void DrawStr(Font font, String str)
{
    //Инициализируем Graphics формы
    Graphics g = this.CreateGraphics();
    //Генерируем случайные координаты, куда будет
    //выведена строка
    int x = random.Next(10, 200);
    int y = random.Next(10, 400);
    //Выводим строку на форму
    g.DrawString(str, font, sldBrush, x, y);
}
```

Добавим на форму два пункта меню **&Font** и **&Clear**.

Пропишем обработчик для первого пункта меню, который позволяет выбрать шрифт для отображаемой строки:

```
private void stmiFont_Click(object sender, EventArgs e)
{
    FontDialog fDialog = new FontDialog();
```

```
    if (fDialog.ShowDialog()== DialogResult.OK)
    {
        Font font = fDialog.Font;
        DrawStr(font, "Строка выведена шрифтом " +
        font.Name);
    }
}
```

При нажатии на **&Clear** производится очистка формы:

```
private void tsmiClear_Click(object sender, EventArgs e)
{
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
}
```

Домашнее задание

Задание 1. Разработать построитель математических графиков функций.

Задание 2. Разработать приложение, создающее диаграммы.