

# Продвинутый С#

Алгоритмы и структуры данных.  
Графический интерфейс.

# Рекурсивные алгоритмы и методы

**Рекурсия** – фундаментальное понятие в математике и компьютерных науках.

**Рекурсивный алгоритм** – алгоритм, который в процессе работы обращается к самому себе.

**Рекурсивный метод** – метод, который вызывает сам себя (то есть реализует рекурсивный алгоритм).

Рекурсивный алгоритм не должен обращаться к себе до бесконечности, следовательно, важная особенность рекурсивного алгоритма – наличие **условия завершения**, позволяющее алгоритму прекратить обращаться к самому себе.

Кроме того, рекурсивный алгоритм должен в процессе работы **достигать** условия завершения.

Работу рекурсивного алгоритма хорошо визуализирует **дерево рекурсии**.

**Метод «разделяй и властвуй»**. Многие алгоритмы используют несколько рекурсивных вызовов (чаще всего два), каждый из которых работает с частью входных данных. Такая рекурсивная схема представляет собой подход **«разделяй и властвуй»** (*divide and conquer*) разработки алгоритмов.

# Условный оператор ?:

Условный оператор (?:) возвращает одно из двух значений в зависимости от значения логического выражения:

условие ? первое\_выражение : второе\_выражение

условие — выражение логического типа (значения только true или false),

если условие имеет значение true, то вычисляется первое\_выражение и результат вычисления является результатом выполнения условного оператора «?:»,

если условие имеет значение false, то вычисляется второе\_выражение и результат вычисления является результатом выполнения условного оператора «?:».

первое\_выражение и второе\_выражение должны быть **одинакового типа** или должно существовать **неявное преобразование** из одного типа в другой.

Например, вместо

```
if (x < 0) result = -1; else result = 1;
```

Можно написать

```
result = x < 0 ? -1 : 1 ;
```

# Структуры

**Структуры** — это **типы значений**, имеющие члены (поля, методы, свойства, конструкторы и т.п.)

Структуры определяются с помощью ключевого слова `struct`.

```
struct StructName
{
    ... //члены структуры
}
```

Например:

```
struct Pixel
{
    int X;
    int Y;
    Color PixelColor; //тип Color из System.Drawing
}
```

В целом структуры используют большую часть того же синтаксиса, что и классы, однако они более ограничены по сравнению с ними.

# Особенности структур

- Структуры являются типами значений, а классы — ссылочными типами.
- В отличие от классов, структуры можно создавать без использования оператора `new`. Однако в этом случае при обращении к полю структуры оно должно быть предварительно инициализировано, а при обращении к структуре в целом (в том числе и при передаче структуры в метод) все поля должны быть предварительно инициализированы.
- В объявлении структуры поля не могут быть инициализированы за исключением констант и статических полей.
- Структура не может иметь пользовательский конструктор без параметров и деструктор. Однако структуры могут иметь конструкторы, имеющие параметры.
- При использовании конструктора без параметров поля структуры заполняются значениями типов по умолчанию (0 для типов значений и `null` для ссылочных типов).
- Структуры могут реализовывать интерфейсы.
- Однако структура не может быть унаследованной от другой структуры или класса и не может быть основой для других классов.

# Особенности структур (окончание)

- Структуры не являются типами, допускающими значение `null`. Однако их можно использовать для создания таких типов. Например:  
`Pixel? pix = null; //объявляется переменная pix типа, допускающего значение //null, который произведен от структуры типа Pixel`
- Структуры копируются при присваивании — выполняется копирование всех данных, а любое изменение новой копии не влияет на данные в исходной копии.
- Структуры копируются при передаче в метод по значению — выполняется копирование всех данных в контекст метода, а любое изменение новой копии не влияет на данные в исходной копии.
- Чтобы изменить исходную структуру в методе, нужно ее передать по ссылке, используя ключевое слово `ref`.
- При апкасте (`upcast`) к типу `object` (упаковка — `boxing`) в куче создается объект с полем типа структуры, в которое копируется исходная структура.
- При даункасте (`downcast`) обратно в тип структуры (распаковка — `unboxing`) поле объекта копируется в соответствующую область памяти контекста.

# СПИСКИ

`List<T>` — класс из `System.Collections.Generic`, экземпляры которого представляют собой собрание элементов типа `T`, доступных по индексу. В отличие от массивов, количество элементов списка не задается и может изменяться в процессе работы программы.

Списки можно задавать так:

```
var intList = new List<int> {2, 5, 0}; // список с целыми числами 2, 5, 0
var strList = new List<string>(); // пустой список строк
```

**Основные свойства:** `Count` (число элементов в списке), `Capacity` (число зарезервированных элементов для хранения списка в памяти)

**Основные методы:**

`Add(elem)` — добавить элемент `elem` в конец списка

`Insert(i, elem)` — вставка элемента `elem` по индексу `i`

`Remove(elem)` — удаляет первое вхождение элемента `elem` из списка

`RemoveAt(i)` — удаляет элемент списка, находящийся по индексу `i`

`Clear()` — удаляет из списка все элементы

К элементам списка можно обращаться по индексу  
`var n = intList[0] + intList[1];`

# Словари

`Dictionary<TKey, TValue>` — класс, экземпляры которого представляют коллекцию пар ключ-значение типов `TKey` и `TValue` соответственно.

Каждый ключ в словаре должен быть уникальным. Ключ не может быть `null`, но значение может быть, если тип значения `TValue` является ссылочным типом.

Задать словарь можно так:

```
var dict = new Dictionary<string, int>();
```

**Основные свойства:** `Count` (число элементов), `Keys` (коллекция ключей), `Values` (коллекция значений).

**Основные методы:**

`Add(key, val)` — добавляет значение `val` по указанному ключу `key`

`ContainsKey(key)` — определяет, содержится ли указанный ключ в словаре

`ContainsValue(val)` — определяет, содержится ли указанное значение в словаре

`Remove(key)` — удаляет элемент словаря по ключу `key`

`Clear()` — удаляет все элементы словаря

К элементам словаря можно обращаться по ключу, например:

```
dict["abc"] = 5; //работает как dict.Add("abc", 5), если ключа "abc" нет или  
//изменяет значение по ключу "abc" на 5, если такой ключ есть
```



# Стеки

`Stack<T>` — класс, экземпляры которого представляют коллекцию переменного размера элементов типа `T`, обслуживаемую по принципу «последним пришел — первым вышел» (LIFO).

Стеки можно задавать так:

```
var strStack = new Stack<string>(); // пустой стек строк
var intStack = new Stack<int> (new int[] {2, 5, 0}); // стек с целыми
// числами 2, 5, 0
```

Стек имеет **свойство** `Count` — число элементов в стеке.

## Основные методы:

`Push(elem)` — добавляет элемент `elem` в стек

`Pop()` — возвращает последний добавленный элемент, удаляя его из стека.

Если стек пуст, то этот метод вызовет ошибку выполнения `InvalidOperationException`

`Peek()` — возвращает последний добавленный элемент, но не удаляет его из стека

`Contains(elem)` — определяет, входит ли элемент `elem` в стек

`Clear()` — удаляет все элементы из стека

# Очереди

Queue<T> — класс, экземпляры которого представляют коллекцию переменного размера элементов типа T, обслуживаемую по принципу «первым пришел — первым вышел» (FIFO).

Очереди можно задавать так:

```
var strQ = new Queue<string>(); //пустая очередь из строк
var intQ = new Queue<int>(new int[] {1, 0, 2}); //очередь целых чисел,
//в которую последовательно помещены элементы массива
```

Очередь имеет **свойство Count** — число элементов в очереди.

Основные методы:

Enqueue(elem) — добавляет элемент elem в конец очереди

Dequeue() — возвращает первый элемент очереди и удаляет его из очереди.  
Если очередь пуста, то этот метод вызовет ошибку выполнения  
InvalidOperationException

Peek() — возвращает первый элемент очереди, но не удаляет его.

Contains(elem) — определяет, входит ли элемент elem в очередь

Clear() — удаляет все элементы из очереди

# Дженерики — универсальные или обобщенные классы

Дженерики инкапсулируют операции, не относящиеся к какому-либо определенному типу данных.

Примерами дженериков могут служить изученные ранее коллекции такие, как списки, словари, стеки и очереди из `System.Collection.Generic`.

Такие операции как добавление элементов в коллекцию или их удаление осуществляются одинаково вне зависимости от типа хранящихся данных.

При создании собственных дженериков следует учитывать следующее:

- какие типы преобразовывать в параметры,
- какие ограничения применять к параметрам типов,
- следует ли разделять поведение дженерика на базовые классы и подклассы,
- следует ли реализовывать один или несколько универсальных интерфейсов.

Пример:

```
public class MyGenericClass<T>
{
    public T Data {get; set;}
    public string Description {get; set;}
}
```

Здесь `T` — параметр типа. Свойство `Data` хранит значение (или объект) типа `T`.

# Ограничения параметров типа

При определении дженерика можно ограничить виды типов, которые могут использоваться в качестве аргументов типа.

При попытке создать экземпляр класса с помощью типа, который не допускается ограничением возникает ошибка компиляции.

Ограничения определяются с помощью ключевого слова `where`

`where T: struct` — T должен иметь **тип значения**. Допускается указание любого типа значения, кроме `Nullable`.

`where T : class` — T должен иметь **ссылочный тип**; это также распространяется на тип любого класса, интерфейса, делегата или массива.

`where T : new()` — T должен иметь **открытый конструктор без параметров**. При использовании с другими ограничениями ограничение `new()` должно устанавливаться последним.

`where T : <base class>` — T **являться или быть производным** от указанно базового класса.

`where T : <interface>` — T должен **являться или реализовывать указанный интерфейс**. Можно установить несколько ограничений интерфейса. Ограничивающий интерфейс также может быть универсальным.

`where T : U` — T должен совпадать с параметром типа U, или быть производным от него.

# Делегаты

**Делегат** - это тип, который представляет собой ссылку на метод с определенным списком параметров и возвращаемым типом.

Делегат определяется при помощи ключевого слова `delegate`. Например:

```
public delegate int UnaryOperation(int x);
```

При создании экземпляра делегата этот экземпляр можно связать с любым методом с совместимой сигнатурой и возвращаемым типом. Впоследствии этот метод можно вызвать с помощью экземпляра делегата.

Делегаты используются для определения **методов обратного вызова** (то есть когда исполняемый код передается в качестве одного из параметров другого кода), в частности, для передачи методов в качестве аргументов к другим методам.

**Основные свойства делегата** (как типа):

`Method` — указатель на метод

`Target` — указатель на объект для динамического метода

Точное соответствие методов типу делегата не требуется (**вариативность**):

- можно создавать экземпляр делегата с методом, тип возвращаемого значения которого неявно наследуют от типа возвращаемого значения в сигнатуре делегата (**ковариация**).
- можно создавать экземпляр делегата с методом, если типы параметров в сигнатуре делегата неявно наследуют от типов соответствующих параметров этого метода (**контрвариация**).

# Универсальные делегаты (дженерик-делегаты)

Можно создавать дженерик-делегаты, используя в их определении параметры типа. Например:

```
public delegate T UnaryOperation<T>(T x);
```

Можно не придумывать собственные делегаты, а воспользоваться имеющимися в .NET дженерик-делегатами `Func` и `Action`

`Func<T1, T2, ... , TN, TResult>` — метод, принимающий  $n$  аргументов типов `T1, T2, ..., TN` (где  $N = 0, 1, ...$ ) и возвращающий значение типа `TResult`

`Action<T1, T2, ... , TN>` — метод, принимающий  $n$  аргументов типов `T1, T2, ..., TN` (где  $N = 0, 1, ...$ ) и не возвращающий значение

Например для определения метода обратного вызова `Map` вместо

```
public delegate T UnaryOperation<T>(T x);  
static T Map<T>(T[] array, UnaryOperation<T> op) {...}
```

можно написать

```
static T Map<T>(T[] array, Func<T, T> op) {...}
```

# Анонимные делегаты и лямбда-выражения

Если делегат нужен только для того, чтобы передать исполняемый код в метод обратного вызова, можно в явном виде не определять метод, на основе которого будет создан экземпляр делегата, а воспользоваться синтаксисом анонимного делегата:

```
delegate(T1 параметр1, T2 параметр2, ..., TN параметрN)
{
    .... // код метода
    return выражение; //выражение, принимающее значение типа TResult
}
```

Чтобы еще больше упростить код (особенно, если метод состоит из одной строки кода) можно использовать лямбда-выражения:

(параметр1, параметр2, ..., параметрN) => выражение

Однако, лямбда-выражения допускают и несколько строчек кода:

```
(параметр1, параметр2, ..., параметрN) =>
{
    .... // код метода
    return выражение;
}
```

Если параметр один, его можно не заключать в скобки:

параметр => выражение

Лямбда-выражение может быть и без параметров:

() => выражение

# Замыкание

Предположим, что в методе определен анонимный делегат.

**Внешняя переменная** (outer variable) — это локальная переменная или параметр (за исключением ref- и out-параметров), доступные в контексте метода. (Внешней она является относительно делегата, так как декларируется вне его.)

Ситуация, когда код анонимного делегата использует внешние переменные, называется **замыканием** (closure), а такие переменные называются **захваченными внешними переменными** (captured outer variable) или просто **захваченными переменными** (captured variable).

При захвате переменной создается объект анонимного типа, поле которого связывается с внешней переменной. Таким образом происходит захват самой внешней переменной, а не ее значения на момент создания экземпляра делегата.

Использование замыкания может привести к интуитивно непонятому поведению кода (так называемая «ловушка замыкания»).

Чтобы избежать «ловушки замыкания», не следует захватывать переменные циклов.



# Взаимодействие «источник—наблюдатель» и мультикаст-делегаты

Взаимодействие объектов программы часто организуется по принципу «источник—наблюдатель». То есть одни объекты генерируют некие события, а другие наблюдают за этими событиями и если событие произошло, то реагируют на него.

То есть одни объекты являются **источником** (source) события, а другой объект — **наблюдателем** (observer).

Взаимодействие «источник—наблюдатель» реализуется на основе делегатов.

Для того, чтобы была возможность реагирование на одно событие нескольких объектов, в современном C# тип делегата образуются как производный от класса `System.MulticastDelegate` (а не от его родительского класса `System.Delegate`), в котором помимо традиционного свойства `Method` инкапсулировано непубличное свойство `_invocationList`, содержащее список вызываемых делегатом методов. Тем самым каждый делегат является многоадресным делегатом (мультикаст-делегатом) .

Извне доступ к этому списку модно получить через метод `GetInvocationList`.

Добавить метод в этот список можно при помощи операции `+=`

Удалить метод из этого списка можно при помощи операции `-=`

Мультикаст-делегат, возвращающий значение, возвращает значение последнего вызванного метода.

# События

**Событие** — это член класса, который позволяет компактно реализовать взаимодействие «источник-наблюдатель».

Объекты-наблюдатели «регистрируют» методы- обработчики события, которые срабатывают, когда событие происходит, то есть когда вызывается специальный метод источника.

По сути событие — это синтаксическая «обертка» над частным делегатным полем, реализующая (явно или неявно) два метода `add` и `remove`.

Событие определяется с помощью ключевого слова **event**:

[модификаторы] `event` тип имя\_события;

где тип определяет тип делегатного поля, то есть сигнатуру методов-обработчиков.

# Соглашение по оформлению событий

Несмотря на то, событие может быть оформлено достаточно произвольно, Microsoft предложило конвенцию (соглашение) по оформлению событий. В частности, все события, встречающиеся в .NET соблюдают эту конвенцию.

Эта конвенция раньше была актуальна, теперь в связи с появлением в языке обобщенных делегатов и лямбда-выражений она потеряла свою актуальность, но многие программисты ее до сих пор придерживаются.

Вот основные положения конвенции:

- информация о событии должна быть инкапсулирована в объект типа `EventArgs` или производного от него с именем, имеющим суффикс `EventArgs`;
- делегатный тип события должен иметь уникальное имя с суффиксом `EventHandler` и иметь два параметра — один типа `object` (задает источник события), другой — типа информации о событии (см. выше);
- метод вызова обработчиков в источнике должен иметь модификаторы `protected virtual`, иметь имя `OnИмя_события`, параметр — информация о событии (как второй параметр в сигнатуре делегата) и перед вызовом обработчиков проверять событие на `null`.

Можно не определять собственный делегат, а воспользоваться имеющимися делегатами `EventHandler` или `EventHandler<T>`.

# Создание Windows-приложений с графическим интерфейсом

Что такое **API**?

Это **application programming interface** (интерфейс программирования приложений) — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах.

**Windows Forms** — часть .NET Framework.

Windows Forms — API, отвечающий за графический интерфейс пользователя, то есть основа Windows-приложений.

**Окна** (windows) — то, что видит и с чем работает пользователь.

**Формы** (forms) — это окна в процессе разработки.

Внутри .NET Framework Windows Forms реализуются в рамках пространства имён System.Windows.Forms

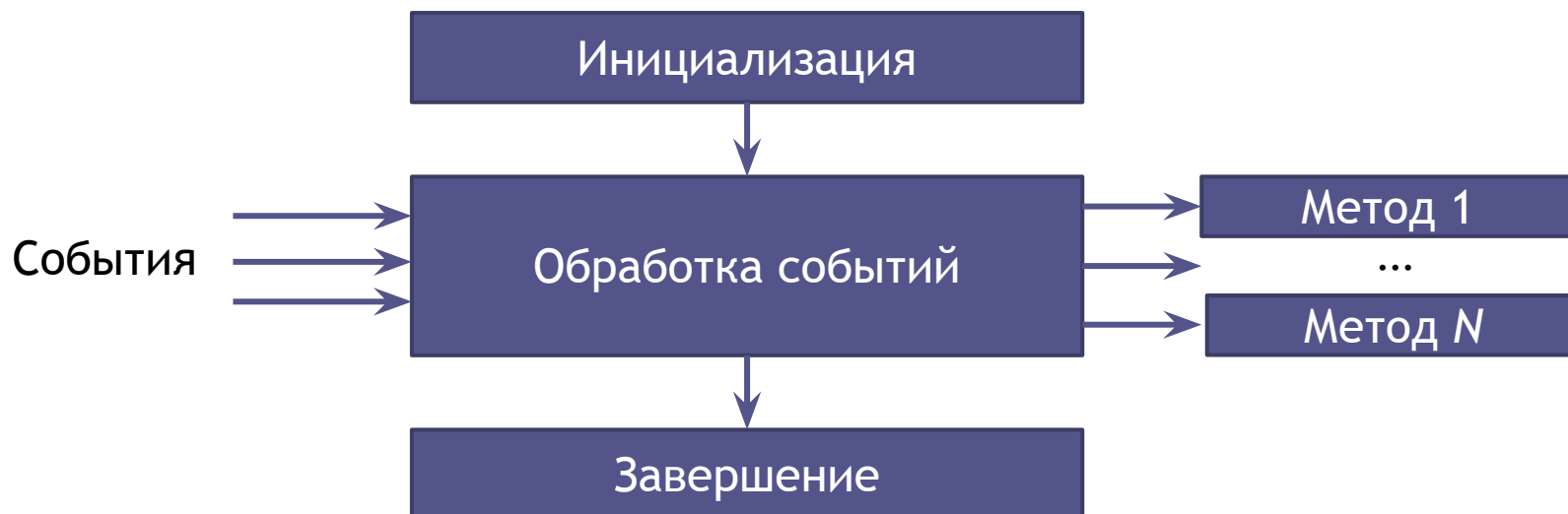
Все формы, используемые в программах, являются экземплярами класса Windows.Forms.Form

# Событийно-управляемое программирование

Основной принцип создания Windows-приложений — **событийно-управляемое программирование**.

Операционная система Windows и запущенное приложение ожидают определенных событий (например, действий пользователя) и реагируют на них определенным образом.

Структура событийно-управляемой программы:



# Процесс создания Windows-приложения

- **Визуальное проектирование.** Создание графического интерфейса приложения.
- **Событийное программирование.** Определение «поведения» программы путем программирования методов обработки событий.

Графический интерфейс программы, разрабатываемый при помощи .NET Framework Windows Forms, основан на создании форм, размещении на них элементов управления и определении свойств форм и этих элементов.

Разработку графического интерфейса можно осуществлять двумя способами:

- создание кода, определяющего формы и элементы управления интерфейса;
- использование визуального средства проектирования — редактора форм, встроенного в Visual Studio. В этом случае код создается автоматически средой разработки.

Разработку Windows-приложения с помощью Windows Forms рекомендуется начинать с шаблона Windows Forms Application.

# Свойства форм

Основные группы свойств:

- **Appearance** — внешний вид объект
- **Behavior** — поведение объекта
- **Design** — состояние объекта во время разработки приложения
- **Layout** — формат объекта
- **Misc** — различные свойства
- **Window style** — стиль окна

# Отображение и скрытие форм

Стартовая форма отображается при запуске приложений.

Чтобы отобразить форму (как обычную форму), нужно использовать метод `Show()`

Чтобы временно скрыть форму, используйте ее свойство `Visible`.

Чтобы закрыть форму (и удалить ее из оперативной памяти), используют метод `Close()`

Чтобы обратиться к форме из самого класса, можно использовать имя `this`. Например, для закрытия формы методом класса этой формы:  
`this.Close();`

Чтобы исключить возможность перехода к другой форме, можно использовать **модальные формы** — эти формы активны, пока они не закрыты, при этом остальные формы приложения остаются неактивными.

Модальные формы отображаются методом `ShowDialog()`

Если модальную форму используют для диалогового окна, ей задают

- неизменяемые размеры (`FormBorderStyle = FixedDialog`)
- убирают кнопки восстановления и свертывания (`MaximizeBox = false, MinimizeBox = false`)
- на форме размещают кнопку подтверждения ввода (например, ОК) и отмены ввода (например, Cancel) с установкой свойств `AcceptedButton = имя_кнопки_ОК` и `CancelButton = имя_кнопки_Cancel`
- у этих кнопок устанавливают соответствующее значение свойства `DialogResult`



# MDI-формы

Интерфейс приложения, организованный при помощи форм бывает двух видов:

- SDI (Single Document Interface) — иерархия форм отсутствует, все формы «равны» между собой.
- MDI (Multiple Document Interface) — есть иерархия форм: одна *родительская форма* (контейнер) и *дочерние* формы, окна которых будут располагаться внутри окна родительской формы.

Для организации родительского окна значение свойства `IsMdiContainer` формы следует сделать равным `True`.

У дочерних форм нужно установить значение свойства `MdiParent` так, чтобы оно указывало на родительскую форму.

# Основные элементы управления

Элементы управления (controls) — основные уже разработанные компоненты интерфейса программ, которые можно использовать в своих приложениях.

Они располагаются на специальной панели инструментов (Toolbox).

## Ввод и отображение текстовой информации

**Label** (метка) — для отображения статичного текста.

Основное свойство — `Text`.

Полезное — `TextAlign` в сочетании с `AutoSize` со значением `False`.

**TextBox** (текстовое поле) — для отображения, ввода и редактирования текста.

Основное свойство — `Text`.

Полезные — `Multiline`, `ScrollBars`, `MaxLength`, `PasswordChar`.

**RichTextBox** (поле форматированного текста) — для ввода текста в формате RTF.

Для загрузки текста (в формате TXT или RTF) в такое поле из файла можно использовать метод `LoadFile`, для сохранения — `SaveFile`.

# Кнопки, флажки, переключатели и списки

**Button** (кнопка) .

Основное свойство — `Text` (надпись на кнопке).

Основное событие — `Click`.

Свойства формы `AcceptButton` и `CancelButton` могут назначать кнопки, которым будет посылаться событие `Click` при нажатии клавиши `Enter` или `ESC` соответственно.

**CheckBox** (флажок) — для графического отображения булевых значений. Основные свойства — `Checked` (выбран ли) и `Text` (надпись около флажка).

**RadioButton** (переключатель) — тоже для графического отображения булевых значений, но из группы переключателей выбран может быть только один.

Чтобы создать несколько групп переключателей, их нужно объединять в контейнеры.

**ListBox** (список) — для отображения списка и выбора его элементов.

Основные свойства — `Items` (коллекция элементов списка), `SelectionMode`, `SelectedItem` или `SelectedIndex`, `Sorted`.

**ComboBox** (комбинированный список) — объединяет элементы `TextBox` и `ListBox`.

Полезное свойство — `DropDownStyle`.

# Элементы управления: контейнер изображения и индикатор прогресса

**Контейнер изображения** (PictureBox) — представляет элемент управления графическим окном для отображения рисунка.

Основные свойства:

**Image** — возвращает или задаёт изображение (объект), отображаемое элементом управления PictureBox .

**ImageLocation** — возвращает или задаёт путь или URL-адрес изображения, отображаемого в PictureBox.

Основной метод:

**Load** — загружает изображение, указанное в свойстве ImageLocation.

**Индикатор прогресса** (ProgressBar) — графически отображает свое значение (применяется для отображения процесса выполнения какой-либо задачи).

Основные свойства:

**Minimum** — минимальное значение.

**Maximum** — максимальное значение.

**Value** — текущее значение.

**Step** — задает шаг изменения значения.

Основной метод:

**PerformStep** — изменяет значение индикатора на величину, заданную свойством Step

# Неотображаемые элементы управления: таймер и диалоговые окна открытия или сохранения файла

**Таймер (Timer)** — элемент управления, вызывающий через определенный интервал времени событие Tick.

Основные свойства:

**Enabled** — логического типа, определяет является ли таймер включенным (True) или выключенным (False).

**Interval** — определяет, через какое время будет генерироваться очередное событие Tick (в миллисекундах).

**Диалоговое окно открытия файла (OpenFileDialog)** — служит для отображения стандартного окна открытия файла.

**Диалоговое окно сохранения файла (SaveFileDialog)** — служит для отображения стандартного окна сохранения файла.

Основные свойства:

**InitialDirectory** — начальный каталог, который будет показан диалоговым окном.

**Filter** — строка фильтра имен файлов, определяет типы файлов, которые будут отображаться в диалоговом окне

**MultiSelect** — позволяет (True) или запрещает (False) выбор нескольких файлов.

**FileName** — полная спецификация выбранного файла (путь + полное имя).

**SafeFileName** — полное имя выбранного файла (имя.расширение).

# События клавиатуры

Полный список событий элементов управления можно найти в документации по ссылке [http://msdn.microsoft.com/ru-ru/library/system.windows.forms.control\\_events\(v=vs.110\).aspx](http://msdn.microsoft.com/ru-ru/library/system.windows.forms.control_events(v=vs.110).aspx)

События нажатия клавиши происходят в следующем порядке.

- PreviewKeyDown
- KeyDown
- KeyPress
- KeyUp

**PreviewKeyDown** — генерируется перед событием KeyDown при нажатии любой клавиши, в том числе управляющей (Enter, ESC, Del и т.п.), модифицирующей (Ctrl, Alt, Shift) или клавиши навигации (стрелки, Home, End и т.п.). В обработчике этого события вы можете задать свойство `e.IsInputKey = True` чтобы вызывать событие KeyDown для такой клавиши. Для обработки считывания или модификации символа нажатой клавиши используется свойство `e.KeyCode`

**KeyDown** — генерируется в первоначальный момент нажатия клавиши. Вызывается нажатием любой клавиши, для которой метод `IsInputKey` возвращает значение True.

Для анализа нажатия модифицирующих клавиш используются свойства `e.Alt`, `e.Control` или `e.Shift`

**KeyPress** — генерируется любой клавишей с символом. Для обработки считывания или модификации символа нажатой клавиши используется свойство `e.KeyChar`

**KeyUp** — генерируется в момент отпускания клавиши. Вызывается отпусканием любой клавиши, для которой метод `IsInputKey` возвращает значение True.

Чтобы обрабатывать события клавиатуры только на уровне формы без предоставления другим элементам управления возможности получать события клавиатуры, необходимо задать для свойства `e.Handled` в методе обработки события KeyPress формы значение True

Для обработки событий клавиатуры следует использовать делегаты типа `PreviewKeyDownEventHandler`, `KeyEventHandler`, `KeyPressEventHandler` (или производных от них типов), и информацию о событии типа `PreviewKeyDownEventArgs`, `EventArgs`, `KeyPressEventArgs` соответственно (или производных от них типов).

# Основы работы с графикой

Для рисования линий, геометрических фигур и текста нужен экземпляр класса `Graphics` из `System.Drawing`.

В этом классе нет публичных конструкторов, так что такой объект нельзя создать при помощи оператора `new`.

Три способа создания объекта типа `Graphics`:

- получить ссылку на объект из параметра `PaintEventArgs`, передаваемого в обработчик события `Paint`, которое возникает всякий раз при необходимости перерисовки формы или элемента управления, например:

```
private void MyForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
}
```
- использование метода `CreateGraphics`, определенного в классе формы и элемента управления, например:

```
Graphics g = this.CreateGraphics();
```
- для возможности рисования на растровом изображении использование статического метода `FromImage` класса `Graphics`, например:

```
Bitmap pic = new Bitmap("c:\Pictures\photo.jpg");
Graphics g = Graphics.FromImage(pic);
```

# Графика (продолжение)

После создания объекта типа `Graphics` его можно использовать для рисования.

Основные классы, экземпляры которых полезны для рисования:

- `Pen` (перо — для рисования линий и контуров)
- `Brush` (кисть — для закрашивания областей)
- `Font` (шрифт — для вывода текста)
- `Color` (цвет)

Процесс рисования сводится к последовательному применению к объекту типа `Graphics` методов `Draw...` или `Fill...`, например:

```
g.DrawLine(new Pen(Color.Red, 10), 80, 4, 200, 200); // рисуем красную линию
g.FillEllipse(Brushes.Blue, 10, 20, 80, 80); // рисуем синий круг
```

Для перерисовки окна при изменении размеров в конструкторе формы следует установить значение `ResizeRedraw` равным `true`:

```
SetStyle(ControlStyles.ResizeRedraw, true);
```

или в обработчике события `SizeChanged` вызвать метод `Refresh()`.

Для предотвращения «мерцания» формы или элемента управления рекомендуется включать двойную буферизацию отрисовки, установив значение `DoubleBuffered` равными `true`:

```
this.DoubleBuffered = true;
```

или

```
SetStyle(ControlStyles.DoubleBuffer, true);
```

Наряду с этим рекомендуется установить значение `true` для `AllPaintingInWmPaint`:  

```
SetStyle(ControlStyles.AllPaintingInWmPaint, true);
```



# Графика (окончание)

Графические объекты потребляют системные ресурсы, поэтому рекомендуется их удалять из памяти сразу, как только они перестают быть необходимыми, не дожидаясь сборки мусора. Для этого нужно вызывать метод `Dispose()`.

Например,

```
Pen p = new Pen(Color.Red, 10); // выделяем ресурс (создаем объект типа Pen)
g.DrawLine(p, 80, 4, 200, 200); // рисуем линию
g.DrawRectangle(p, 10, 10, 150, 250); // рисуем прямоугольник
p.Dispose(); // освобождаем ресурс
```

Для упрощения можно использовать синтаксис `using`:

`using` (выделение\_ресурса) оператор

Например:

```
using ( Pen p = new Pen(Color.Red, 10) )
{
    g.DrawLine(p, 80, 4, 200, 200);
    g.DrawRectangle(p, 10, 10, 150, 250);
}
```