**Ответы на контрольные вопросы**

**№ 2**

Какое основное назначение технологии Windows Forms, ее особенности, преимущества и недостатки?

Windows Forms — это платформа пользовательского интерфейса для создания классических приложений Windows.

Она обеспечивает один из самых эффективных способов создания классических приложений с помощью визуального конструктора в Visual Studio. Такие функции, как размещение визуальных элементов управления путем перетаскивания, упрощают создание классических приложений.

Преимущества

Он не отвечает за отрисовку окон, меню и т.п., эта задача закреплена за пользовательской подсистемой, располагающейся в user32.dll и основывающейся на GDI. Одно из преимуществ Windows. Forms - в том, что на нем можно писать коссплатформенные приложения. Простые проекты, написанные на Windows. Forms, можно довольно легко перенести на другую операционную систему, если на ней установлен.

Недостатки

Минусы: Писать xaml гораздо дольше, чем пользоваться дизайнером Windows Forms. Кроме того, декларативный код и большей концентрации требует при написании, и отлаживать его практически невозможно. Ну и в целом архитектура технологии не располагает к быстрой разработке, сложнее автоматизировать рутину. Непредсказуемое поведение при изменении темы Windows.

Зачем используется класс Form? Назовите основные методы, свойства и события данного класса.

Представляет окно или диалоговое окно, которое составляет пользовательский интерфейс приложения.

Свойства

AcceptButton - Читает или устанавливает информацию о кнопке, которая будет "нажата" (в форме), когда пользователь нажмет клавишу ‹Enter›.

ActiveMDIChild, IsMDIChild, IsMDIContainer - Используются в контексте МDI-приложения

CancelButton - Читает или устанавливает информацию о кнопочном элементе управления, который будет "нажат", когда пользователь нажмет клавишу ‹Esc›

ControlBox - Читает или устанавливает значение, являющееся индикатором наличия у формы экранной кнопки управления окном FormBorderStyle - Читает или устанавливает значение, задающее стиль границы формы (в соответствии с перечнем FormBorderStyle)

Menu - Читает или устанавливает информацию о стыковке меню в форме MaximizeBox

MinimizeBox - Используются для информации о наличии у формы кнопок минимизации и максимизации окна ShowInTaskbar

ShowInTaskbar - Указывает, будет ли форма видимой в панели задач Windows

StartPosition - Читает или устанавливает значение, задающее начальную позицию окна формы (в соответствии с перечнем FormStartPosition) WindowState - Указывает (в соответствии с перечнем FormWindowState), в каком виде должна отображаться форма при запуске

Метод

Activate() - Активизирует форму и предоставляет ей фокус ввода

Close() - Закрывает форму

CenterToScreen() - Размещает форму в центре экрана

LayoutMDI - Размещает все дочерние формы (в соответствии с перечнем LayoutMDI) в рамках родительской формы

ShowDialog() - Отображает форму в виде модального диалогового окна.

События

Activated - Происходит при активизации формы, т.е. при получении формой фокуса ввода Closed.

Closing Используются для проверки того, что форма закрывается или уже закрыта

Deactivate Происходит при деактивизации формы, те. когда форма утрачивает текущий фокус ввода

Load Происходит после того, как форма размещается в памяти, но пока остается невидимой на экране

MDIChildActive Генерируется при активизации дочернего окна

Поясните структуру проекта и назначение всех файлов?

Внешний вид приложения является нам преимущественно через формы. Формы являются основными строительными блоками. Они предоставляют контейнер для различных элементов управления. А механизм событий позволяет элементам формы отзываться на ввод пользователя, и, таким образом, взаимодействовать с пользователем.

При открытии проекта в Visual Studio в графическом редакторе мы можем увидеть визуальную часть формы - ту часть, которую мы видим после запуска приложения и куда мы переносим элементы с панели управления. Но на самом деле форма скрывает мощный функционал, состоящий из методов, свойств, событий и прочее. Рассмотрим основные свойства форм.

Если мы запустим приложение, то нам отобразится одна пустая форма. Однако даже такой простой проект с пустой формой имеет несколько компонентов:

Несмотря на то, что мы видим только форму, но стартовой точкой входа в графическое приложение является класс Program, расположенный в файле Program.cs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using System.Windows.Forms;    namespace HelloApp  {      static class Program      {          [STAThread]          static void Main()          {              Application.EnableVisualStyles();              Application.SetCompatibleTextRenderingDefault(false);              Application.Run(new Form1());          }      }  } |

Сначала программой запускается данный класс, затем с помощью выражения Application.Run(new Form1()) он запускает форму Form1. Если вдруг мы захотим изменить стартовую форму в приложении на какую-нибудь другую, то нам надо изменить в этом выражении Form1 на соответствующий класс формы.

Сама форма сложна по содержанию. Она делится на ряд компонентов. Так, в структуре проекта есть файл Form1.Designer.cs, который выглядит примерно так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | namespace HelloApp  {      partial class Form1      {          /// <summary>          /// Required designer variable.          /// </summary>          private System.ComponentModel.IContainer components = null;            /// <summary>          /// Clean up any resources being used.          /// </summary>          /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>          protected override void Dispose(bool disposing)          {              if (disposing && (components != null))              {                  components.Dispose();              }              base.Dispose(disposing);          }            #region Windows Form Designer generated code            /// <summary>          /// Required method for Designer support - do not modify          /// the contents of this method with the code editor.          /// </summary>          private void InitializeComponent()          {              this.SuspendLayout();              //              // Form1              //              this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);              this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;              this.ClientSize = new System.Drawing.Size(284, 261);              this.Name = "Form1";              this.Text = "Привет мир!";              this.ResumeLayout(false);            }            #endregion        }  } |

Здесь объявляется частичный класс формы Form1, которая имеет два метода: Dispose(), который выполняет роль деструктора объекта, и InitializeComponent(), который устанавливает начальные значения свойств формы.

При добавлении элементов управления, например, кнопок, их описание также добавляется в этот файл.

Но на практике мы редко будем сталкиваться с этим классом, так как они выполняет в основном дизайнерские функции - установка свойств объектов, установка переменных.

Еще один файл - Form1.resx - хранит ресурсы формы. Как правило, ресурсы используются для создания однообразных форм сразу для нескольких языковых культур.

И более важный файл - Form1.cs, который в структуре проекта называется просто Form1, содержит код или программную логику формы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using System;  using System.Collections.Generic;  using System.ComponentModel;  using System.Data;  using System.Drawing;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.Windows.Forms;    namespace HelloApp  {      public partial class Form1 : Form      {          public Form1()          {              InitializeComponent();          }      }  } |

По умолчанию здесь есть только конструктор формы, в котором просто вызывается метод InitializeComponent(), объявленный в файле дизайнера Form1.Designer.cs. Именно с этим файлом мы и будем больше работать.

4. Зачем нужен атрибут STAThreadAttribute?

Атрибут STAThreadAttribute указывает, что модель потока COM для приложения является однопоточным. Этот атрибут должен присутствовать в точке входа любого приложения, использующего Windows Forms; если он опущен, компоненты Windows могут работать неправильно.

5. Как в вашем проекте используются события и делегаты?

Объявление делегата определяет класс, производный от класса System.Delegate. Экземпляр делегата инкапсулирует список вызовов, который представляет собой список одного или нескольких методов, каждый из которых называется вызываемой сущностью. Для методов экземпляра вызываемая сущность состоит из экземпляра и метода в этом экземпляре. Для статических методов вызываемая сущность состоит только из метода. Вызов экземпляра делегата с соответствующим набором аргументов приводит к тому, что каждая из вызываемых сущностей экземпляра делегата вызывается с заданным набором аргументов.

Для взаимодействия с пользователем в Windows Forms используется механизм событий. События в Windows Forms представляют стандартные события на C#, только применяемые к визуальным компонентам и подчиняются тем же правилам, что события в C#. Но создание обработчиков событий в Windows Forms все же имеет некоторые особенности.

Прежде всего в WinForms есть некоторый стандартный набор событий, который по большей части имеется у всех визуальных компонентов. Отдельные элементы добавляют свои события, но принципы работы с ними будут похожие. Чтобы посмотреть все события элемента, нам надо выбрать этот элемент в поле графического дизайнера и перейти к вкладке событий на панели форм. Например, события формы:

Чтобы добавить обработчик, можно просто два раза нажать по пустому полю рядом с названием события, и после этого Visual Studio автоматически сгенерирует обработчик события. Например, нажмем для создания обработчика для события Load:

И в этом поле отобразится название метода обработчика события Load. По умолчанию он называется Form1\_Load.

Если мы перейдем в файл кода формы Form1.cs, то увидим автосгенерированный метод Form1\_Load:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public partial class Form1 : Form  {      public Form1()      {          InitializeComponent();      }        private void Form1\_Load(object sender, EventArgs e)      {        }  } |

И при каждой загрузке формы будет срабатывать код в обработчике Form1\_Load.

Как правило, большинство обработчиков различных визуальных компонентов имеют два параметра: sender - объект, инициировавший событие, и аргумент, хранящий информацию о событии (в данном случае EventArgs e).

Но это только обработчик. Добавление же обработчика, созданного таким образом, производится в файле Form1.Designer.cs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | namespace HelloApp  {      partial class Form1      {          private System.ComponentModel.IContainer components = null;            protected override void Dispose(bool disposing)          {              if (disposing && (components != null))              {                  components.Dispose();              }              base.Dispose(disposing);          }          private void InitializeComponent()          {              this.SuspendLayout();                this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);              this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;              this.ClientSize = new System.Drawing.Size(284, 261);              this.Name = "Form1";              // добавление обработчика              this.Load += new System.EventHandler(this.Form1\_Load);              this.ResumeLayout(false);          }      }  } |

Для добавления обработчика используется стандартный синтаксис C#: this.Load += new System.EventHandler(this.Form1\_Load)

Поэтому если мы захотим удалить созданный подобным образом обработчик, то нам надо не только удалить метод из кода формы в Form1.cs, но и удалить добавление обработчика в этом файле.

Однако мы можем добавлять обработчики событий и програмно, например, в конструкторе формы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | using System;  using System.Collections.Generic;  using System.ComponentModel;  using System.Data;  using System.Drawing;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.Windows.Forms;    namespace HelloApp  {      public partial class Form1 : Form      {          public Form1()          {              InitializeComponent();              this.Load += LoadEvent;          }            private void Form1\_Load(object sender, EventArgs e)          {          }            private void LoadEvent(object sender, EventArgs e)          {              this.BackColor = Color.Yellow;          }      }  } |

Кроме ранее созданного обработчика Form1\_Load здесь также добавлен другой обработчик загрузки формы: this.Load += LoadEvent;, который устанавливает в качестве фона желтый цвет.

6. Объясните схему работы цепочек делегатов.

Цепочка делегатов позволяет создавать связанный список делегатов, так что когда вызывается делегат, находящийся в начале списка, вслед за ним выполняются все делегаты цепочки.

7. Объясните механизм подписки и отмены подписки на события.

Подписка на события программными средствами

Определите метод обработчика событий, сигнатура которого соответствует сигнатуре делегата для события. Например, если событие основано на типе делегата [EventHandler](https://docs.microsoft.com/ru-ru/dotnet/api/system.eventhandler), то следующий код представляет заглушку метода:

C#Копировать

void HandleCustomEvent(object sender, CustomEventArgs a)

{

// Do something useful here.

}

Чтобы присоединить обработчик событий к событию, используйте оператор присваивания сложения (+=). В приведенном ниже примере предположим, что объект с именем publisher имеет событие с именем RaiseCustomEvent. Обратите внимание на то, что классу подписчика требуется ссылка на класс издателя, чтобы подписаться на его события.

C#Копировать

publisher.RaiseCustomEvent += HandleCustomEvent;

Чтобы указать обработчик событий, можно также воспользоваться [лямбда-выражением](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/lambda-expressions):

C#Копировать

public Form1()

{

InitializeComponent();

this.Click += (s,e) =>

{

MessageBox.Show(((MouseEventArgs)e).Location.ToString());

};

}

Подписка на события с помощью анонимной функции

Если вам не нужно будет позже отменять подписку на событие, можно использовать оператор присваивания сложения (+=) для привязки анонимной функции как обработчика событий. В следующем примере предположим, что объект с именем publisher имеет событие с именем RaiseCustomEvent и что класс CustomEventArgs также был определен и содержит некие относящиеся к событию сведения. Обратите внимание на то, что классу подписчика требуется ссылка на publisher, чтобы подписаться на его события.

C#Копировать

publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>

{

string s = o.ToString() + " " + e.ToString();

Console.WriteLine(s);

};

Отменить подписку на событие не так просто, если для подписки использовалась анонимная функция. Чтобы отменить подписку в этом случае, в коде, где была выполнена подписка на событие, сохраните анонимную функцию в переменной делегата, а затем добавьте делегат к событию. Мы рекомендуем не использовать анонимные функции для подписки на события, если в будущем вам нужно будет отменить подписку. Дополнительные сведения об анонимных функциях см. в статье [Лямбда-выражения](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/lambda-expressions).

Отмена подписки

Чтобы предотвратить вызов обработчика событий при инициировании события, подписку на событие необходимо отменить. Во избежание утечки ресурсов отменять подписку на события следует до удаления объекта подписчика. До тех пор пока подписка на событие не отменена, делегат многоадресной рассылки, лежащий в основе события в публикующем объекте, будет ссылаться на делегат, инкапсулирующий обработчик событий подписчика. Если ссылка присутствует в публикующем объекте, объект подписчика не будет удален при сборке мусора.

Отмена подписки на событие

Чтобы отменить подписку на событие, воспользуйтесь оператором присваивания вычитания (-=).

C#Копировать

publisher.RaiseCustomEvent -= HandleCustomEvent;

Если подписка на событие отменена для всех подписчиков, экземпляр события в классе издателя получает значение null.

8. Как создать вторую форму и передать туда данные? Есть ли другие способы?

Чтобы добавить еще одну форму в проект, нажмем на имя проекта в окне Solution Explorer (Обозреватель решений) правой кнопкой мыши и выберем Add(Добавить)->Windows Form... .

Дадим новой форме какое-нибудь имя, например, *Form2.cs*:

Итак, у нас в проект была добавлена вторая форма. Теперь попробуем осуществить взаимодействие между двумя формами. Допустим, первая форма по нажатию на кнопку будет вызывать вторую форму. Во-первых, добавим на первую форму Form1 кнопку и двойным щелчком по кнопке перейдем в файл кода. Итак, мы попадем в обработчик события нажатия кнопки, который создается по умолчанию после двойного щелчка по кнопке:

|  |  |
| --- | --- |
| 1  2  3  4 | private void button1\_Click(object sender, EventArgs e)  {    } |

Теперь добавим в него код вызова второй формы. У нас вторая форма называется Form2, поэтому сначала мы создаем объект данного класса, а потом для его отображения на экране вызываем метод Show:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private void button1\_Click(object sender, EventArgs e)  {      Form2 newForm = new Form2();      newForm.Show();  } |

Теперь сделаем наоборот - чтобы вторая форма воздействовала на первую. Пока вторая форма не знает о существовании первой. Чтобы это исправить, надо второй форме как-то передать сведения о первой форме. Для этого воспользуемся передачей ссылки на форму в конструкторе.

Итак перейдем ко второй форме и перейдем к ее коду - нажмем правой кнопкой мыши на форму и выберем View Code (Просмотр кода). Пока он пустой и содержит только конструктор. Поскольку C# поддерживает перегрузку методов, то мы можем создать несколько методов и конструкторов с разными параметрами и в зависимости от ситуации вызывать один из них. Итак, изменим файл кода второй формы на следующий:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | using System;  using System.Collections.Generic;  using System.ComponentModel;  using System.Data;  using System.Drawing;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.Windows.Forms;    namespace HelloApp  {      public partial class Form2 : Form      {          public Form2()          {              InitializeComponent();          }            public Form2(Form1 f)          {              InitializeComponent();              f.BackColor = Color.Yellow;          }      }  } |

Фактически мы только добавили здесь новый конструктор public Form2(Form1 f), в котором мы получаем первую форму и устанавливаем ее фон в желтый цвет. Теперь перейдем к коду первой формы, где мы вызывали вторую форму и изменим его на следующий:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private void button1\_Click(object sender, EventArgs e)  {      Form2 newForm = new Form2(this);      newForm.Show();  } |

Поскольку в данном случае ключевое слово this представляет ссылку на текущий объект - объект Form1, то при создании второй формы она будет получать ее (ссылку) и через нее управлять первой формой.

Теперь после нажатия на кнопку у нас будет создана вторая форма, которая сразу изменит цвет первой формы.

Мы можем также создавать объекты и текущей формы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | private void button1\_Click(object sender, EventArgs e)  {      Form1 newForm1 = new Form1();      newForm1.Show();        Form2 newForm2 = new Form2(newForm1);      newForm2.Show();  } |

При работе с несколькими формами надо учитывать, что одна из них является главной - которая запускается первой в файле Program.cs. Если у нас одновременно открыта куча форм, то при закрытии главной закрывается все приложение и вместе с ним все остальные формы.

9. Как во время выполнения приложения добавить/удалить элемент управления?

**Программное добавление элемента управления в коллекцию**

Создайте экземпляр элемента управления, подлежащий добавлению.

Задайте свойства нового элемента управления.

Добавьте этот элемент управления в коллекцию Controls родительского элемента управления.

В следующем примере кода демонстрируется создание экземпляра элемента управления [Button](https://docs.microsoft.com/ru-RU/dotnet/api/system.windows.forms.button). Необходима форма с элементом управления NewPanelButton\_Click, уже должен существовать метод обработки событий для создаваемой кнопки [Panel](https://docs.microsoft.com/ru-RU/dotnet/api/system.windows.forms.panel).

C#Копировать

public Button newPanelButton = new Button();

public void addNewControl()

{

// The Add method will accept as a parameter any object that derives

// from the Control class. In this case, it is a Button control.

panel1.Controls.Add(newPanelButton);

// The event handler indicated for the Click event in the code

// below is used as an example. Substitute the appropriate event

// handler for your application.

this.newPanelButton.Click += new System.EventHandler(this. NewPanelButton\_Click);

}

**Программное удаление элементов управления из коллекции**

Удалите обработчик событий из события. В Visual Basic используйте ключевое слово [оператор RemoveHandler](https://docs.microsoft.com/ru-RU/dotnet/visual-basic/language-reference/statements/removehandler-statement), в C# используйте [оператор -=](https://docs.microsoft.com/ru-RU/dotnet/csharp/language-reference/operators/subtraction-operator).

Используйте метод Remove для удаления требуемого элемента управления из коллекции Controls панели.

Вызовите метод [Dispose](https://docs.microsoft.com/ru-RU/dotnet/api/system.windows.forms.control.dispose), чтобы освободить все ресурсы, используемые элементом управления.

C#Копировать

private void removeControl(object sender, System.EventArgs e)

{

// NOTE: The code below uses the instance of

// the button (newPanelButton) from the previous example.

if(panel1.Controls.Contains(newPanelButton))

{

this.newPanelButton.Click -= new System.EventHandler(this.

NewPanelButton\_Click);

panel1.Controls.Remove(newPanelButton);

newPanelButton.Dispose();

}

}

**№ 3**

1. Назовите классы, которые используются для создания меню. Перечислите свойства и методы.

класс [MenuItem](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menuitem?view=netframework-4.6), [ContextMenu](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.contextmenu?view=netframework-4.6), ClaseMenu и MergeMenu

Свойства

|  |  |
| --- | --- |
| [CanRaiseEvents](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component.canraiseevents?view=netframework-4.6#system-componentmodel-component-canraiseevents) | Возвращает значение, показывающее, может ли компонент вызывать событие. |
| [Container](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component.container?view=netframework-4.6#system-componentmodel-component-container) | Возвращает объект [IContainer](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.icontainer?view=netframework-4.6), который содержит коллекцию [Component](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component?view=netframework-4.6). |
| [DesignMode](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component.designmode?view=netframework-4.6#system-componentmodel-component-designmode) | Возвращает значение, указывающее, находится ли данный компонент [Component](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component?view=netframework-4.6) в режиме конструктора в настоящее время. |
| [Events](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component.events?view=netframework-4.6#system-componentmodel-component-events) | Возвращает список обработчиков событий, которые прикреплены к этому объекту [Component](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component?view=netframework-4.6). |
| [Handle](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.handle?view=netframework-4.6#system-windows-forms-menu-handle) | Получает значение, представляющее дескриптор окна для меню. |
| [IsParent](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.isparent?view=netframework-4.6#system-windows-forms-menu-isparent) | Получает значение, показывающее, содержит ли это меню какие-нибудь элементы меню. Это свойство доступно только для чтения. |
| [MdiListItem](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.mdilistitem?view=netframework-4.6#system-windows-forms-menu-mdilistitem) | Получает значение, указывающее объект [MenuItem](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menuitem?view=netframework-4.6), используемый для отображения списка дочерних форм многодокументного интерфейса (MDI). |
| [MenuItems](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.menuitems?view=netframework-4.6#system-windows-forms-menu-menuitems) | Получает значение, показывающее коллекцию объектов [MenuItem](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menuitem?view=netframework-4.6), связанных с данным меню. |
| [Name](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.name?view=netframework-4.6#system-windows-forms-menu-name) | Возвращает или задает имя таблицы для объекта [Menu](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu?view=netframework-4.6). |
| [Site](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component.site?view=netframework-4.6#system-componentmodel-component-site) | Получает или задает [ISite](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.isite?view=netframework-4.6) объекта [Component](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component?view=netframework-4.6). |
| [Tag](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.tag?view=netframework-4.6#system-windows-forms-menu-tag) | Получает или задает определяемые пользователем данные, связанные с элементом управления. |

Методы

|  |  |
| --- | --- |
| [CloneMenu(Menu)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.clonemenu?view=netframework-4.6#system-windows-forms-menu-clonemenu(system-windows-forms-menu)) | Копирует объект [Menu](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu?view=netframework-4.6), передаваемый в качестве параметра в текущий объект [Menu](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu?view=netframework-4.6). |
| [CreateMenuHandle()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.createmenuhandle?view=netframework-4.6#system-windows-forms-menu-createmenuhandle) | Создает новый дескриптор для объекта [Menu](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu?view=netframework-4.6). |
| [CreateObjRef(Type)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.marshalbyrefobject.createobjref?view=netframework-4.6#system-marshalbyrefobject-createobjref(system-type)) | Создает объект, который содержит всю необходимую информацию для создания прокси-сервера, используемого для взаимодействия с удаленным объектом. |
| [Dispose()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component.dispose?view=netframework-4.6#system-componentmodel-component-dispose) | Освобождает все ресурсы, занятые модулем [Component](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component?view=netframework-4.6). |
| [Dispose(Boolean)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.dispose?view=netframework-4.6#system-windows-forms-menu-dispose(system-boolean)) | Уничтожает ресурсы (кроме памяти), используемые классом [Menu](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu?view=netframework-4.6). |
| [Equals(Object)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.object.equals?view=netframework-4.6#system-object-equals(system-object)) | Определяет, равен ли указанный объект текущему объекту. |
| [FindMenuItem(Int32, IntPtr)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.findmenuitem?view=netframework-4.6#system-windows-forms-menu-findmenuitem(system-int32-system-intptr)) | Получает объект [MenuItem](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menuitem?view=netframework-4.6), содержащий указываемое значение. |
| [FindMergePosition(Int32)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.findmergeposition?view=netframework-4.6#system-windows-forms-menu-findmergeposition(system-int32)) | Возвращает позицию меню, в которую должен быть вставлен элемент меню. |
| [GetContextMenu()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.getcontextmenu?view=netframework-4.6#system-windows-forms-menu-getcontextmenu) | Получает объект [ContextMenu](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.contextmenu?view=netframework-4.6), содержащий это меню. |
| [GetHashCode()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.object.gethashcode?view=netframework-4.6#system-object-gethashcode) | Служит хэш-функцией по умолчанию.  (Унаследовано от [Object](https://docs.microsoft.com/ru-Ru/dotnet/api/system.object?view=netframework-4.6)) |
| [GetLifetimeService()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.marshalbyrefobject.getlifetimeservice?view=netframework-4.6#system-marshalbyrefobject-getlifetimeservice) | Извлекает объект обслуживания во время существования, который управляет политикой времени существования данного экземпляра.  (Унаследовано от [MarshalByRefObject](https://docs.microsoft.com/ru-Ru/dotnet/api/system.marshalbyrefobject?view=netframework-4.6)) |
| [GetMainMenu()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.getmainmenu?view=netframework-4.6#system-windows-forms-menu-getmainmenu) | Получает объект [MainMenu](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.mainmenu?view=netframework-4.6), содержащий это меню. |
| [GetService(Type)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component.getservice?view=netframework-4.6#system-componentmodel-component-getservice(system-type)) | Возвращает объект, представляющий службу, предоставляемую классом [Component](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.component?view=netframework-4.6) или классом [Container](https://docs.microsoft.com/ru-Ru/dotnet/api/system.componentmodel.container?view=netframework-4.6). |
| [GetType()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.object.gettype?view=netframework-4.6#system-object-gettype) | Возвращает объект [Type](https://docs.microsoft.com/ru-Ru/dotnet/api/system.type?view=netframework-4.6) для текущего экземпляра. |
| [InitializeLifetimeService()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.marshalbyrefobject.initializelifetimeservice?view=netframework-4.6#system-marshalbyrefobject-initializelifetimeservice) | Получает объект службы времени существования для управления политикой времени существования для этого экземпляра. |
| [MemberwiseClone()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.object.memberwiseclone?view=netframework-4.6#system-object-memberwiseclone) | Создает неполную копию текущего объекта [Object](https://docs.microsoft.com/ru-Ru/dotnet/api/system.object?view=netframework-4.6). |
| [MemberwiseClone(Boolean)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.marshalbyrefobject.memberwiseclone?view=netframework-4.6#system-marshalbyrefobject-memberwiseclone(system-boolean)) | Создает неполную копию текущего объекта [MarshalByRefObject](https://docs.microsoft.com/ru-Ru/dotnet/api/system.marshalbyrefobject?view=netframework-4.6). |
| [MergeMenu(Menu)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.mergemenu?view=netframework-4.6#system-windows-forms-menu-mergemenu(system-windows-forms-menu)) | Объединяет объекты [MenuItem](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menuitem?view=netframework-4.6) одного меню с текущим меню. |
| [ProcessCmdKey(Message, Keys)](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.processcmdkey?view=netframework-4.6#system-windows-forms-menu-processcmdkey(system-windows-forms-message@-system-windows-forms-keys)) | Обрабатывает клавишу для команд. |
| [ToString()](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu.tostring?view=netframework-4.6#system-windows-forms-menu-tostring) | Возвращает объект [String](https://docs.microsoft.com/ru-Ru/dotnet/api/system.string?view=netframework-4.6), который представляет элемент управления [Menu](https://docs.microsoft.com/ru-Ru/dotnet/api/system.windows.forms.menu?view=netframework-4.6). |

2. Что может содержать строка состояния? Какие есть методы управления строкой состояния?

Строка состояния предназначена для отображения текущей информации о состоянии работы приложения.

При добавлении на форму StatusStrip автоматически размещается в нижней части окна приложения (как и в большинстве приложений).

Однако при необходимости мы сможем его иначе позиционировать, управляя свойством Dock, которое может принимать следующие значения:

* Bottom: размещение внизу (значение по умолчанию)
* Top: прикрепляет статусную строку к верхней части формы
* Fill: растягивает на всю форму
* Left: размещение в левой части формы
* Right: размещение в правой части формы
* None: произвольное положение

StatusStrip может содержать различные элементы. В режиме дизайнера мы можем добавить следующие типы элементов:

* StatusLabel: метка для вывода текстовой информации. Представляет объект ToolStripLabel
* ProgressBar: индикатор прогресса. Представляет объект ToolStripProgressBar
* DropDownButton: кнопка с выпадающим списком по клику. Представляет объект ToolStripDropDownButton
* SplitButton: еще одна кнопка, во многом аналогичная DropDownButton. Представляет объект ToolStripSplitButton

Либо можно обратиться на панели свойств к свойству Items компонента StatusStrip и открывшемся окне добавить и настроить все элементы

3. Что такое регулярные выражения? Где и как их можно использовать?

Регулярные выражения предоставляют мощный, гибкий и эффективный способ обработки текста. Комплексная нотация сопоставления шаблонов регулярных выражений позволяет быстро анализировать большие объемы текста в следующих целях:

поиск определенных шаблонов символов;

проверка текста на соответствие предопределенному шаблону (например, адресу электронной почты);

извлечение, изменение, замена или удаление текстовых подстрок;

добавление извлеченных строк в коллекцию для создания отчета.

Для многих приложений, которые работают со строками или анализируют большие блоки текста, регулярные выражения — незаменимый инструмент.

Используют в замене подстрок, поиск повторяющихся слов, динамическое создание регулярного выражения с учетом языка и региональных параметров.

4. Что такое привязки (якоря) в RegEx? Приведите примеры.

Классы StringBuilder и String предоставляют достаточную функциональность для работы со строками. Однако .NET предлагает еще один мощный инструмент - регулярные выражения. Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен **System.Text.RegularExpressions**. А центральным классом при работе с регулярными выражениями является класс **Regex**. Например, у нас есть некоторый текст и нам надо найти в нем все словоформы какого-нибудь слова. С классом Regex это сделать очень просто:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System.Text.RegularExpressions;    string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";  Regex regex = new Regex(@"туп(\w\*)");  MatchCollection matches = regex.Matches(s);  if (matches.Count > 0)  {      foreach (Match match in matches)          Console.WriteLine(match.Value);  }  else  {      Console.WriteLine("Совпадений не найдено");  } |

Здесь мы находим в искомой строке все словоформы слова "туп". В конструктор объекта Regex передается регулярное выражение для поиска. Далее мы разберем некоторые элементы синтаксиса регулярных выражений, а пока достаточно знать, что выражение туп(\w\*) обозначает, найти все слова, которые имеют корень "туп" и после которого может стоять различное количество символов. Выражение \w означает алфавитно-цифровой символ, а звездочка после выражения указывает на неопределенное их количество - их может быть один, два, три или вообще не быть.

Метод Matches класса Regex принимает строку, к которой надо применить регулярные выражения, и возвращает коллекцию найденных совпадений.

Каждый элемент такой коллекции представляет объект **Match**. Его свойство Value возвращает найденное совпадение.

И в данном случае мы получим следующий консольный вывод

тупогуб

тупогубенький

тупа

### Параметр RegexOptions

Класс Regex имеет ряд конструкторов, позволяющих выполнить начальную инициализацию объекта. Две версии конструкторов в качестве одного из параметров принимают перечисление RegexOptions. Некоторые из значений, принимаемых данным перечислением:

* **Compiled**: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
* **CultureInvariant**: при установке этого значения будут игнорироваться региональные различия
* **IgnoreCase**: при установке этого значения будет игнорироваться регистр
* **IgnorePatternWhitespace**: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #
* **Multiline**: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
* **RightToLeft**: приписывает читать строку справа налево
* **Singleline**: при данном режиме символ "." соответствует любому символу, в том числе последовательности "\n", которая осуществляет переход на следующую строку

Например:

|  |  |
| --- | --- |
| 1 | Regex regex = new Regex(@"туп(\w\*)", RegexOptions.IgnoreCase); |

При необходимости можно установить несколько параметров:

|  |  |
| --- | --- |
| 1 | Regex regex = new Regex(@"туп(\w\*)", RegexOptions.Compiled | RegexOptions.IgnoreCase); |

### Синтаксис регулярных выражений

Рассмотрим вкратце некоторые элементы синтаксиса регулярных выражений:

* **^**: соответствие должно начинаться в начале строки (например, выражение @"^пр\w\*" соответствует слову "привет" в строке "привет мир")
* **$**: конец строки (например, выражение @"\w\*ир$" соответствует слову "мир" в строке "привет мир", так как часть "ир" находится в самом конце)
* **.**: знак точки определяет любой одиночный символ (например, выражение "м.р" соответствует слову "мир" или "мор")
* **\***: предыдущий символ повторяется 0 и более раз
* **+**: предыдущий символ повторяется 1 и более раз
* **?**: предыдущий символ повторяется 0 или 1 раз
* **\s**: соответствует любому пробельному символу
* **\S**: соответствует любому символу, не являющемуся пробелом
* **\w**: соответствует любому алфавитно-цифровому символу
* **\W**: соответствует любому не алфавитно-цифровому символу
* **\d**: соответствует любой десятичной цифре
* **\D**: соответствует любому символу, не являющемуся десятичной цифрой

Это только небольшая часть элементов. Более подробное описание синтаксиса регулярных выражений можно найти на msdn в статье [Элементы языка регулярных выражений — краткий справочник](https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference).

Теперь посмотрим на некоторые примеры использования. Возьмем первый пример с скороговоркой "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа" и найдем в ней все слова, где встречается корень "губ":

|  |  |
| --- | --- |
| 1  2 | string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";  Regex regex = new Regex(@"\w\*губ\w\*"); |

Так как выражение \w\* соответствует любой последовательности алфавитно-цифровых символов любой длины, то данное выражение найдет все слова, содержащие корень "губ".

Второй простенький пример - нахождение телефонного номера в формате 111-111-1111:

|  |  |
| --- | --- |
| 1  2 | string s = "456-435-2318";  Regex regex = new Regex(@"\d{3}-\d{3}-\d{4}"); |

Если мы точно знаем, сколько определенных символов должно быть, то мы можем явным образом указать их количество в фигурных скобках: \d{3} - то есть в данном случае три цифры.

Мы можем не только задать поиск по определенным типам символов - пробелы, цифры, но и задать конкретные символы, которые должны входить в регулярное выражение. Например, перепишем пример с номером телефона и явно укажем, какие символы там должны быть:

|  |  |
| --- | --- |
| 1  2 | string s = "456-435-2318";  Regex regex = new Regex("[0-9]{3}-[0-9]{3}-[0-9]{4}"); |

В квадратных скобках задается диапазон символов, которые должны в данном месте встречаться. В итоге данный и предыдущий шаблоны телефонного номера будут эквивалентны.

Также можно задать диапазон для алфавитных символов: Regex regex = new Regex("[a-v]{5}"); - данное выражение будет соответствовать любому сочетанию пяти символов, в котором все символы находятся в диапазоне от a до v.

Можно также указать отдельные значения: Regex regex = new Regex(@"[2]\*-[0-9]{3}-\d{4}");. Это выражение будет соответствовать, например, такому номеру телефона "222-222-2222" (так как первые числа двойки)

С помощью операции | можно задать альтернативные символы, например:

|  |  |
| --- | --- |
| 1 | Regex regex = new Regex(@"(2|3){3}-[0-9]{3}-\d{4}"); |

То есть первые три цифры могут содержать только двойки или тройки. Такой шаблон будет соответствовать, например, строкам "222-222-2222" и "323-435-2318". А вот строка "235-435-2318" уже не подпадает под шаблон, так как одной из трех первых цифр является цифра 5.

Итак, у нас такие символы, как \*, + и ряд других используются в качестве специальных символов. И возникает вопрос, а что делать, если нам надо найти, строки, где содержится точка, звездочка или какой-то другой специальный символ? В этом случае нам надо просто экранировать эти символы слешем:

|  |  |
| --- | --- |
| 1  2 | Regex regex = new Regex(@"(2|3){3}\.[0-9]{3}\.\d{4}");  // этому выражению будет соответствовать строка "222.222.2222" |

### Проверка на соответствие строки формату

Нередко возникает задача проверить корректность данных, введенных пользователем. Это может быть проверка электронного адреса, номера телефона, Класс Regex предоставляет статический метод **IsMatch**, который позволяет проверить входную строку с шаблоном на соответствие:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using System.Text.RegularExpressions;    string pattern = @"^(?("")(""[^""]+?""@)|(([0-9a-z]((\.(?!\.))|[-!#\$%&'\\*\+/=\?\^`\{\}\|~\w])\*)(?<=[0-9a-z])@))" +                  @"(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-z][-\w]\*[0-9a-z]\*\.)+[a-z0-9]{2,17}))$";  var data = new string[]  {      "tom@gmail.com",      "+12345678999",      "bob@yahoo.com",      "+13435465566",      "sam@yandex.ru",      "+43743989393"  };    Console.WriteLine("Email List");  for(int i = 0; i < data.Length; i++)  {      if (Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase))      {          Console.WriteLine(data[i]);      }  } |

Переменная pattern задает регулярное выражение для проверки адреса электронной почты. Данное выражение предлагает нам Microsoft на страницах msdn.

Далее в цикле мы проходим по массиву строк и определяем, какие строки соответствуют этому шаблону, то есть представляют валидный адрес электронной почты. Для проверки соответствия строки шаблону используется метод IsMatch: Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase). Последний параметр указывает, что регистр можно игнорировать. И если строка соответствует шаблону, то метод возвращает true.

### Замена и метод Replace

Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string text = "Мама  мыла  раму. ";  string pattern = @"\s+";  string target = " ";  Regex regex = new Regex(pattern);  string result = regex.Replace(text, target);  Console.WriteLine(result); |

Данная версия метода Replace принимает два параметра: строку с текстом, где надо выполнить замену, и сама строка замены. Так как в качестве шаблона выбрано выражение "\s+ (то есть наличие одного и более пробелов), метод Replace проходит по всему тексту и заменяет несколько подряд идущих пробелов ординарными.

Другой пример - на вход подается номер телефона в произвольном формате, и мы хотим оставить в нем только цифры:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string phoneNumber = "+1(876)-234-12-98";  string pattern = @"\D";  string target = "";  Regex regex = new Regex(pattern);  string result = regex.Replace(phoneNumber, target);  Console.WriteLine(result);  // 18762341298 |

В данном случае шаблон "\D" представляет любой символ, который не является цифрой. Любой такой символ заменяется на пустую строку "", то есть в итоге из строки "+1(876)-234-12-98" мы получим строку "18762341298".

5. Зачем используют конструкции группирования? Приведите примеры.

**Конструкции** **группирования** отображают части выражений регулярных выражений и захватывают части строки входной строки. **Конструкции** **группирования** можно **использовать** в следующих целях. Сопоставление с частью выражения, которая повторяется во входной строке. Применение квантификатору к части выражения с несколькими языковыми элементами регулярных выражений.

GroupBox представляет собой специальный контейнер, который ограничен от остальной формы границей. Он имеет заголовок, который устанавливается через свойство Text. Чтобы сделать GroupBox без заголовка, в качестве значения свойства Text просто устанавливается пустая строка.

Нередко этот элемент используется для группирования переключателей - элементов RadioButton, так как позволяет разграничить их группы.

Элемент Panel представляет панель и также, как и GroupBox, объединяет элементы в группы. Она может визуально сливаться с остальной формой, если она имеет то же значение цвета фона в свойстве BackColor, что и форма. Чтобы ее выделить можно кроме цвета указать для элемента границы с помощью свойства BorderStyle, которое по умолчанию имеет значение None, то есть отсутствие границ.

Также если панель имеет много элементов, которые выходят за ее границы, мы можем сделать прокручиваемую панель, установив ее свойство AutoScroll в true

Также, как и форма, GroupBox и Panel имеют коллекции элементов, и мы также можем динамически добавлять в эти контейнеры элементы. Например, на форме есть элемент GroupBox, который имеет имя groupBox1:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | private void Form1\_Load(object sender, EventArgs e)  {      Button helloButton = new Button();      helloButton.BackColor = Color.LightGray;      helloButton.ForeColor = Color.Red;      helloButton.Location = new Point(30, 30);      helloButton.Text = "Привет";      groupBox1.Controls.Add(helloButton);  } |

Для указания расположения элемента в контейнере мы используем структуру Point: new Point(30, 30);, которой в конструкторе передаем размещение по осям Х и Y. Эти координаты устанавливаются относительно левого верхнего угла контейнера - то есть в данном случае элемента GroupBox

При этом надо учитывать, что контейнером верхнего уровня является форма, а элемент groupBox1 сам находится в коллекции элементов формы. И при желании мы могли бы удалить его:

|  |  |
| --- | --- |
| 1 | this.Controls.Remove(groupBox1); |

### FlowLayoutPanel

Элемент FlowLayoutPanel является унаследован от класса Panel, и поэтому наследует все его свойства. Однако при этом добавляя дополнительную функциональность. Так, этот элемент позволяет изменять позиционирование и компоновку дочерних элементов при изменении размеров формы во время выполнения программы.

Свойство элемента **FlowDirection** позволяет задать направление, в котором направлены дочерние элементы. По умолчанию имеет значение LeftToRight - то есть элементы будут располагаться начиная от левого верхнего края. Следующие элементы будут идти вправо. Это свойство также может принимать следующие значения:

* **RightToLeft** - элементы располагаются от правого верхнего угла в левую сторону
* **TopDown** - элементы располагаются от левого верхнего угла и идут вниз
* **BottomUp** - элементы располагаются от левого нижнего угла и идут вверх

При расположении элементов важную роль играет свойство WrapContents. По умолчанию оно имеет значение True. Это позволяет переносить элементы, которые не умещаются в FlowLayoutPanel, на новую строку или в новый столбец. Если оно имеет значение False, то элементы не переносятся, а к контейнеру просто добавляются полосы прокрутки, если свойство AutoScroll равно true.

6. Что такое квантор или множители? Приведите примеры.

Квантификаторы определяют количество экземпляров символа, группы или класса символов, которое должно присутствовать во входных данных, чтобы было зафиксировано совпадение.

Совпадение ноль или несколько раз

string pattern = @"\b91\*9\*\b";

string input = "99 95 919 929 9119 9219 999 9919 91119";

foreach (Match match in Regex.Matches(input, pattern))

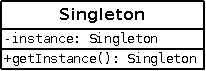
Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

**№4**

1. Какие типы паттернов бывают?

Порождающие - абстрагируют процесс порождения классов и объектов.  
поведения – опр. алгоритмы взаимод. между классами и объектами  
структурные – рассматривает, как классы и объекты образуют более крупные структуры – более сложные по характеру классы и объекты

2. Нарисуете диаграмму классов и поясните принцип работы Singleton. Назовите условия примен.



Используется когда необходимо, чтобы для класса существовал только 1 экземпляр. Позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан.

3. Как сделать потокобезопасную, с отложенной реализацией версию Singleton.

\*Чтобы избежать одновременного доступа к коду из разных потоков критическая секция заключается в блок lock.

\*Lazy-реализация - данные иниц. только перед непосред использованием.

\*Реализация через класс Lazy<T>

4. Нарисуете диаграмму классов и поясните принцип Factory Method. Назовите усл примен

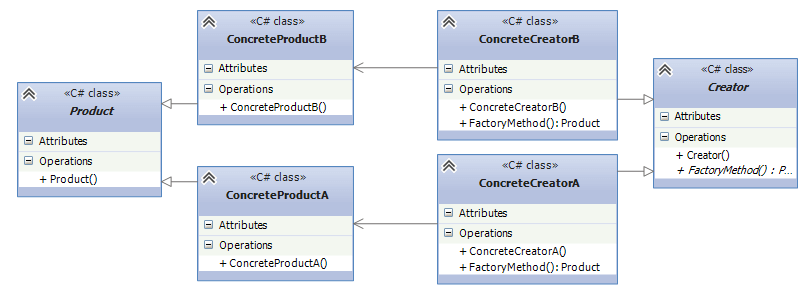
Определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах.

**Когда:**

- заранее известно объекты каких типов необх.создавать

-система дб независимой от процесса создания новых объектов и расширяемой

- создание новых объектов необходимо делегировать из базового класса наследникам.



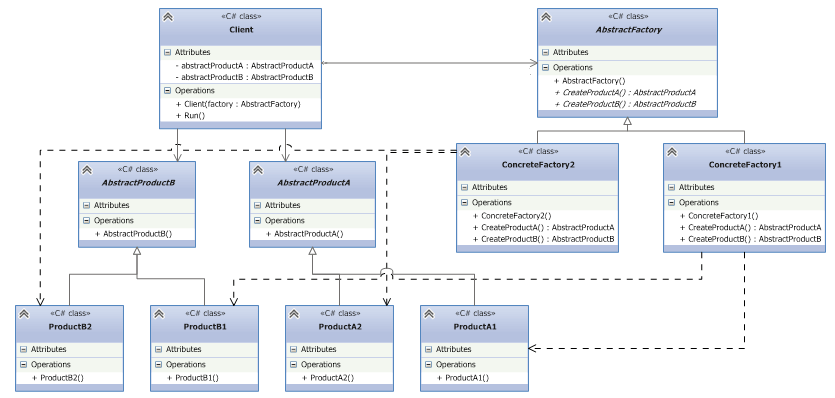
5. Нарисуете диаграмму классов и поясните принцип Abstract Factory + условия применения

Предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов.

**Когда:**

- система не д.зависеть от способа создания и компоновки новых объектов

-создаваемые объекты д.использоваться вместе и явл.взаимосвязанными.



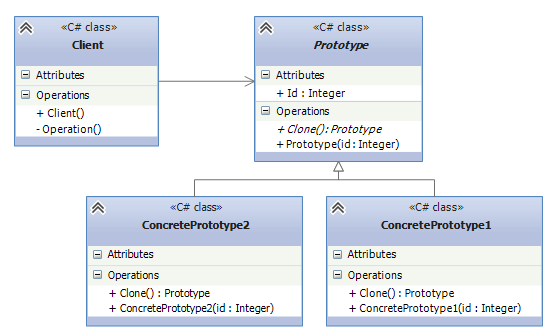
6. Нарисуете диаграмму классов и поясните принцип работы Prototype+ условия применения

Позвол.созд.объекты на осн.ранее созданных объектов-прототипов.(клонирование)

**Когда:** -конкр. тип создаваемого объекта д. определяться динамически во время выполнения

-нежелательно создание отдельной иерархии классов фабрик создания объектов-продуктов из параллельной иерархии классов

- клонирование – более предпочтительный варик нежели создание и инициализирование с помощью конструктора.



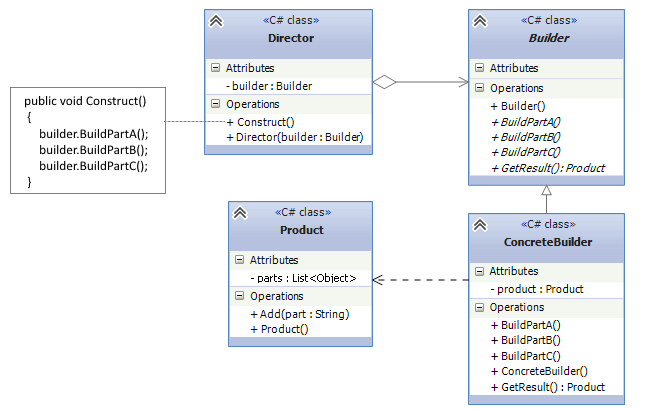
7. Нарисуете диаграмму классов и поясните принцип работ Builder + условия применения

Инкапсулирует создание объекта и позволяет разделить его на различные этапы

**Когда:**

- процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой

- необх. обеспечить получение различных вариаций объекта в процессе его создания



8. Что такое Пул объектов (Object pool), в чем суть паттерна?

Это порождающий паттерн, набор инициализированных и готовых к использованию объектов. Когда системе нужен объект – не создается, а берется из пула. Когда объект больше не нужен – не уничтожается, а возвращается в пул.

Повышает производительность.

9. Какое основное назначение паттерна отложенная инициализация (Lazy initialization)?

Прием, когда некоторая ресурсоемкая операция (создание объекта, вычисление значения) выполняется непосредственно перед тем, как будет использован ее результат.  
инициализация вып. по требованию, а не заблаговременно.

Частный случай – создание объекта в момент обращения к нему – один из порождающих паттернов. Используется в сочетании с фабричным методом, одиночкой и заместителем.

+ускоряется нач.инициал.

+иниц.только тогда, когда необх

-задержка при 1-ом обращ. к объекту

-невозможно явным образом задать порядок инициализации объектов.

ABSTRACT FACTORY:

* Абстрактные классы **AbstractProductA** и **AbstractProductB** определяют интерфейс для классов, объекты которых будут создаваться в программе.
* Конкретные классы **ProductA1 / ProductA2** и **ProductB1 / ProductB2**представляют конкретную реализацию абстрактных классов
* Абстрактный класс фабрики **AbstractFactory** определяет методы для создания объектов. Причем методы возвращают абстрактные продукты, а не их конкретные реализации.
* Конкретные классы фабрик **ConcreteFactory1** и **ConcreteFactory2** реализуют абстрактные методы базового класса и непосредственно определяют какие конкретные продукты использовать
* Класс клиента **Client** использует класс фабрики для создания объектов. При этом он использует исключительно абстрактный класс фабрики AbstractFactory и абстрактные классы продуктов AbstractProductA и AbstractProductB и никак не зависит от их конкретных реализаций

FACTORY METHOD

* Абстрактный класс **Product** определяет интерфейс класса, объекты которого надо создавать.
* Конкретные классы **ConcreteProductA** и **ConcreteProductB** представляют реализацию класса Product. Таких классов может быть множество
* Абстрактный класс **Creator** определяет абстрактный фабричный метод FactoryMethod(), который возвращает объект Product.
* Конкретные классы **ConcreteCreatorA** и **ConcreteCreatorB** - наследники класса Creator, определяющие свою реализацию метода FactoryMethod(). Причем метод FactoryMethod() каждого отдельного класса-создателя возвращает определенный конкретный тип продукта. Для каждого конкретного класса продукта определяется свой конкретный класс создателя.

Таким образом, класс Creator делегирует создание объекта Product своим наследникам. А классы ConcreteCreatorA и ConcreteCreatorB могут самостоятельно выбирать какой конкретный тип продукта им создавать.

PROTOTYPE

* **Prototype**: определяет интерфейс для клонирования самого себя, который, как правило, представляет метод Clone()
* **ConcretePrototype1** и **ConcretePrototype2**: конкретные реализации прототипа. Реализуют метод Clone()
* **Client**: создает объекты прототипов с помощью метода Clone()

BULDER

* **Product**: представляет объект, который должен быть создан. В данном случае все части объекта заключены в списке parts.
* **Builder**: определяет интерфейс для создания различных частей объекта Product
* **ConcreteBuilder**: конкретная реализация Buildera. Создает объект Product и определяет интерфейс для доступа к нему
* **Director**: распорядитель - создает объект, используя объекты Builder

**№5**

1. Назначение структурных паттернов? Рассм.как классы и объекты реализ.более крупные структуры

2. Нарисуете диаграмму классов и поясните принцип работы Adapter. В каких случаях применять.

Предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

**Когда:** - необходимо исп-ть имеющийся класс, но его интерфейс не соотв.потребностям

-надо исп-ть уже сущ-щий класс совместно с другими классами, интерф.кот.несовместимы

3. В чем разница между адаптером класса и адаптером объекта?

Адаптер класса исп.наследование и может переносить только класс . Он не может обернуть интерфейс, поскольку по определению он должен выводиться из некоторого базового класса.  
Адаптер объектов исп.состав и может переносить классы или интерфейсы или и то, и другое. Он может это сделать, т.к.он сод. как закрытый, инкапсулир.член, класс или интерф. экземпляр объекта, кот.он обертывает.

4. Назначение и принцип организации паттерна Декоратор.

Это структурный шаблон проектирования, который позволяет динамически подключать к объекту доп. функциональность.

Для определения нового функционала в классах исп-ся наследование, а декоратор представляет более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов.

**Когда:** надо динам.добавлять к объекту новые функциональные возм-сти (д-е мб сняты с объекта)

5. Нарисуйте диаграмму классов для паттерна Proxy. Поясните его назначение + разновидности

Заместитель предоставляет объект-заместитель, который управляет доступом к другом объекту.

То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его.

**Когда:** - надо осущ.взаимодействие по сети, а объект-проси должен имитировать поведение объекта в другом адресном пространстве. Использование похволит снизить накладные издежки при передаче данных через сеть. Подобная ситуация называется **удалённый заместитель**

**-** нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобиться, а до этого все запросы к нему обрабатывает прокси-объект. Ситуация – **виртуальный заместитель.**

**-** необходимо разграничить доступ к вызываемому объекту в завис-ти от прав вызывающего объекта. Ситуация – **защищающий заместитель**

**-** нужно вести подсчет ссылок на объект или обеспечить потокобезопасную работу с реальным объектом. Ситуация – **умные ссылки**

6. В чем разница между паттернами Decorator и Adapter?

**Адаптер** и **Декоратор** во многом похожи, однако при реализации **Адаптера у** вас нет новой логики, кроме преобразования. При реализации **Decorator** вы фактически добавляете некоторые совершенно новые функциональные возможности, которые никогда не существовали ранее в объекте, который вы украшаете

7. В чем суть паттерна Composite? Поясните в каких случаях его надо применять?

Компоновщик объединяет группы объектов в древовидную структуру по принципу «часть-целое и позволяет клиенту работать как с отдельными объектами, так и с группой объектов.

Реализация – в виде меню, которое имеет различные пункты ( м.содерж.подменю, кот.также им.пункты)

**Когда:** - объекты дб реализованы в виде иерархической древовидной структуры

- клиенты единообразно должны управлять как целыми объектами, так и их составными частями. Т.е. целое и его части д. реализовать один и тот же интерфейс.

8. Расскажите о паттерне Façade?

Позвол.скрыть сложность системы с пом.представления упрощенного интерфейса для взаимодейсвтия с ней

**Когда: -** имеется сложн.сист., и необх.упростить с ней работу. Позвол.опред.одну точку взаимодействия между клиентом и системой.

- надо уменьшить кол-во завис-тей между клиентом и сложн.сист. Объекты позвол.отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.

- нужно опред.подсист.компонентов в сложн.сист. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимод.между ними и повысить их независ-ть друг от друга.

9. В каких случаях надо применять Bridge? Поясните на диаграмме классов как он применятся

Мост – структ.паттерн, кот.позвол.отделить абстракцию от реализации таким образом, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга. Даже если отделить абстр.от реализ., то наслед.классы все равно будут жестко привязаны к интерфейсу, опред-мому в баз.абстр.классе.

**Когда:** - надо избежать пост.привязки абстракции к реализации

- наряду с реализ.надо измен.и абстракцию независ. друг от друга.(измен.в абстр.!= измен. в реализ)

**ADAPTER**

class Client

{

    public void Request(Target target)

    {

        target.Request();

    }

}

// класс, к которому надо адаптировать другой класс

class Target

{

    public virtual void Request()

    {}

}

// Адаптер

class Adapter : Target

{

    private Adaptee adaptee = new Adaptee();

    public override void Request()

    {

        adaptee.SpecificRequest();

    }

}

// Адаптируемый класс

class Adaptee

{

    public void SpecificRequest()

    {}

}

* **Target**: представляет объекты, которые используются клиентом
* **Client**: использует объекты Target для реализации своих задач
* **Adaptee**: представляет адаптируемый класс, который мы хотели бы использовать у клиента вместо объектов Target
* **Adapter**: собственно адаптер, который позволяет работать с объектами Adaptee как с объектами Target.

То есть клиент ничего не знает об Adaptee, он знает и использует только объекты Target. И благодаря адаптеру мы можем на клиенте использовать объекты Adaptee как Target

**Decorator (декоратор)**

abstract class Component

{

    public abstract void Operation();

}

class ConcreteComponent : Component

{

    public override void Operation(){}

}

abstract class Decorator : Component

{

    protected Component component;

    public void SetComponent(Component component)

    {this.component = component; }

    public override void Operation()

    {

        if (component != null)

            component.Operation();

    }

}

class ConcreteDecoratorA : Decorator

{

    public override void Operation()

    {base.Operation(); }

}

class ConcreteDecoratorB : Decorator

{

    public override void Operation()

    {base.Operation(); }

}

* **Component**: абстрактный класс, который определяет интерфейс для наследуемых объектов
* **ConcreteComponent**: конкретная реализация компонента, в которую с помощью декоратора добавляется новая функциональность
* **Decorator**: собственно декоратор, реализуется в виде абстрактного класса и имеет тот же базовый класс, что и декорируемые объекты. Поэтому базовый класс Component должен быть по возможности легким и определять только базовый интерфейс.

Класс декоратора также хранит ссылку на декорируемый объект в виде объекта базового класса Component и реализует связь с базовым классом как через наследование, так и через отношение агрегации.

* Классы **ConcreteDecoratorA** и **ConcreteDecoratorB** представляют дополнительные функциональности, которыми должен быть расширен объект ConcreteComponent.

PROXY

class Client

{

    void Main()

    {

        Subject subject = new Proxy();

        subject.Request();

    }

}

abstract class Subject

{

    public abstract void Request();

}

class RealSubject : Subject

{

    public override void Request()

    {}

}

class Proxy : Subject

{

    RealSubject realSubject;

    public override void Request()

    {

        if (realSubject == null)

            realSubject = new RealSubject();

        realSubject.Request();

    }

}

* **Subject**: определяет общий интерфейс для Proxy и RealSubject. Поэтому Proxy может использоваться вместо RealSubject
* **RealSubject**: представляет реальный объект, для которого создается прокси
* **Proxy**: заместитель реального объекта. Хранит ссылку на реальный объект, контролирует к нему доступ, может управлять его созданием и удалением. При необходимости Proxy переадресует запросы объекту RealSubject
* **Client**: использует объект Proxy для доступа к объекту RealSubject

COMPOSITE(КОМПОНОВЩИК)

abstract class Component

{

    protected string name;

    public Component(string name)

    {

        this.name = name;

    }

    public abstract void Display();

    public abstract void Add(Component c);

    public abstract void Remove(Component c);

}

class Composite : Component

{

    List<Component> children = new List<Component>();

    public Composite(string name)

        : base(name)

    {}

    public override void Add(Component component)

    {

        children.Add(component);

    }

    public override void Remove(Component component)

    {

        children.Remove(component);

    }

    public override void Display()

    {

        Console.WriteLine(name);

        foreach (Component component in children)

        {

            component.Display();

        }

    }

}

class Leaf : Component

{

    public Leaf(string name)

        : base(name)

    {}

    public override void Display()

    {

        Console.WriteLine(name);

    }

    public override void Add(Component component)

    {

        throw new NotImplementedException();

    }

    public override void Remove(Component component)

    {

        throw new NotImplementedException();

    }

}

* **Component**: определяет интерфейс для всех компонентов в древовидной структуре
* **Composite**: представляет компонент, который может содержать другие компоненты и реализует механизм для их добавления и удаления
* **Leaf**: представляет отдельный компонент, который не может содержать другие компоненты
* **Client**: клиент, который использует компоненты

FAÇADE(ФАСАД)

class SubsystemA

{ public void A1() {} }

class SubsystemB

{ public void B1() {} }

class SubsystemC

{ public void C1()  {} }

public class Facade

{

    SubsystemA subsystemA;

    SubsystemB subsystemB;

    SubsystemC subsystemC;

    public Facade(SubsystemA sa, SubsystemB sb, SubsystemC sc)

    {

        subsystemA = sa;

        subsystemB = sb;

        subsystemC = sc;

    }

    public void Operation1()

    {

        subsystemA.A1();

        subsystemB.B1();

        subsystemC.C1();

    }

    public void Operation2()

    {

        subsystemB.B1();

        subsystemC.C1();

    }

}

class Client

{

    public void Main()

    {

        Facade facade = new Facade(new SubsystemA(), new SubsystemB(), new SubsystemC());

        facade.Operation1();

        facade.Operation2();

    }

}

* Классы SubsystemA, SubsystemB, SubsystemC и т.д. являются компонентами сложной подсистемы, с которыми должен взаимодействовать клиент
* Client взаимодействует с компонентами подсистемы
* Facade - непосредственно фасад, который предоставляет интерфейс клиенту для работы с компонентами

BRIDGE(МОСТ)

class Client

{

    static void Main() {

        Abstraction abstraction;

        abstraction = new RefinedAbstraction(new ConcreteImplementorA());

        abstraction.Operation();

        abstraction.Implementor=new ConcreteImplementorB();

        abstraction.Operation();

    }

}

abstract class Abstraction

{

    protected Implementor implementor;

public Implementor Implementor

    { set { implementor = value; }    }

public Abstraction(Implementor imp)

    { implementor = imp    }

 public virtual void Operation()

    {   implementor.OperationImp(); }

}

abstract class Implementor

{ public abstract void OperationImp();}

class RefinedAbstraction : Abstraction

{

    public RefinedAbstraction(Implementor imp) : base(imp) {}

    public override void Operation(){}

}

class ConcreteImplementorA : Implementor

{ public override void OperationImp() {} }

class ConcreteImplementorB : Implementor

{ public override void OperationImp() {} }

* **Abstraction**: определяет базовый интерфейс и хранит ссылку на объект Implementor. Выполнение операций в Abstraction делегируется методам объекта Implementor
* **RefinedAbstraction**: уточненная абстракция, наследуется от Abstraction и расширяет унаследованный интерфейс
* **Implementor**: определяет базовый интерфейс для конкретных реализаций. Как правило, Implementor определяет только примитивные операции. Более сложные операции, которые базируются на примитивных, определяются в Abstraction
* **ConcreteImplementorA** и **ConcreteImplementorB**: конкретные реализации, которые унаследованы от Implementor
* **Client**: использует объекты Abstraction

1. Назначение паттернов поведения?

Паттерны, опред-щие алгоритмы и взаимодействие между классами и объектами

2. Принцип работы Chain of responsibility + применять?

Цепочка обязанностей  
позвол. избежать жесткой привязки отправителя запроса к получателю, позволяя нескольким объектам обработать запрос.  
запрос перемещается по цепочке, каждый объект выбирает, либо обработать запрос, либо передать следующему.

3. Назначение и принцип организации паттерна Command. Поясните как он связан с конечными автоматами

Позвол. инкапсулировать запрос на вып-ние опред. действия в виде отдельного объекта  
Этот объект запроса на действие и называется **командой**

**Когда: -** необх. ф-ции обратного действия в ответ на опр. действия.

- необх. обесп. вып-ние очереди запросов, а также их возможную отмену.

4. Как реализовать паттерн Observer?

Исп.отношение один ко многим  
один наблюдаемый объект и мн-во наблюдателей  
При изменении наблюдаемого объекта автомат. происходит оповещение всех наблюдателей.

\* инт1: опр. методы для доб., удал., оповещения наблюдателей  
\* инт2: с пом. кот наблюдатель получ. оповещение   
2 класса : инт 1,2 - один уведомл, другой добавляет

5. Паттерн Mediator. Поясните его назначение.

Обеспеч. взаимод. мн-ва объектов без необх. ссылаться друг на друга  
 -> слабосвязанность взаимод. объектов

6. В чем разница между паттернами Mediator и Facade?

фасад предоставляет существующие функции, а медиатор добавляет к существующим функциям.

7. В чем суть паттерна Memento? Поясните на примере.

Позвол. сохранить состояние объекта для возможного послед. восстановления без нарушения принципа инкапсуляции

8. Расскажите о паттерне Visitor?

доб. один. набор операций разнородным классам без изменения этих классов

9. В каких случаях надо применять Null object? когда требуется инкапсулировать отсутствие объекта.

10. реализовать Strategy.

- неск.родственных классов, кот. отлич.поведением. Созд.1 осн.класс, а разные варики поведения вынести в отдельные классы и при необх.их применять

- необх.обеспеч.выбор из неск.вариков алгоритмов, кот.можно легко менять в завис-ти от условий

- необх.менять поведение объектов на стадии вып-ния программы

11. Перечислите и поясните принципы проектирования SOLID.

Способствует созданию такой системы, кот.будет легко поддерживать и расширять в течении долгого времени. Принципы солид – это руководства, кот.также м.примен.во время работы над сущ. ПО

**CHAIN OF RESPONSIBILITY(ЦЕПОЧКА ОБЯЗАННОСТЕЙ)**

abstract class Handler

{

    public Handler Successor { get; set; }

    public abstract void HandleRequest(int condition);

}

class ConcreteHandler1 : Handler

{

    public override void HandleRequest(int condition)

    {

        if (condition == 1)

        {

            // обработка;

        }

        else if (Successor != null)

        {

            Successor.HandleRequest(condition);

        }

    }

}

class ConcreteHandler2 : Handler

{

    public override void HandleRequest(int condition)

    {

        if (condition==2)

        {

            // обработка;

        }

        else if (Successor != null)

        {

            Successor.HandleRequest(condition);

        }

    }

}

* Handler: определяет интерфейс для обработки запроса. Также может определять ссылку на следующий обработчик запроса
* ConcreteHandler1 и ConcreteHandler2: конкретные обработчики, которые реализуют функционал для обработки запроса. При невозможности обработки и наличия ссылки на следующий обработчик, передают запрос этому обработчику

В данном случае для простоты примера в качестве параметра передается некоторое число, и в зависимости от значения данного числа обработчики и принимают решения об обработке запроса.

* Client: отправляет запрос объекту Handler

**COMMAND (КОМАНДЫ)**

{

    public abstract void Execute();

    public abstract void Undo();

}

// конкретная команда

class ConcreteCommand : Command

{

    Receiver receiver;

    public ConcreteCommand(Receiver r)

    {

        receiver = r;

    }

    public override void Execute()

    {

        receiver.Operation();

    }

    public override void Undo()

    {}

}

// получатель команды

class Receiver

{ public void Operation()  { } }

class Invoker

{

    Command command;

    public void SetCommand(Command c)

    {

        command = c;

    }

    public void Run()

    {   command.Execute();    }

    public void Cancel()

    { command.Undo();   }

}

* **Command**: интерфейс, представляющий команду. Обычно определяет метод Execute() для выполнения действия, а также нередко включает метод Undo(), реализация которого должна заключаться в отмене действия команды
* **ConcreteCommand**: конкретная реализация команды, реализует метод Execute(), в котором вызывается определенный метод, определенный в классе Receiver
* **Receiver**: получатель команды. Определяет действия, которые должны выполняться в результате запроса.
* **Invoker**: инициатор команды - вызывает команду для выполнения определенного запроса
* **Client**: клиент - создает команду и устанавливает ее получателя с помощью метода SetCommand()

**OBSERVER (НАБЛЮДАТЕЛЬ)**

interface IObservable

{

    void AddObserver(IObserver o);

    void RemoveObserver(IObserver o);

    void NotifyObservers();

}

class ConcreteObservable : IObservable

{

    private List<IObserver> observers;

    public ConcreteObservable()

    {

        observers = new List<IObserver>();

    }

    public void AddObserver(IObserver o)

    { observers.Add(o);    }

    public void RemoveObserver(IObserver o)

    { observers.Remove(o);  }

    public void NotifyObservers()

    {

        foreach (IObserver observer in observers)

            observer.Update();

    }

}

interface IObserver

{ void Update(); }

class ConcreteObserver :IObserver

{ public void Update() {} }

* IObservable: представляет наблюдаемый объект. Определяет три метода: AddObserver() (для добавления наблюдателя), RemoveObserver() (удаление набюдателя) и NotifyObservers() (уведомление наблюдателей)
* ConcreteObservable: конкретная реализация интерфейса IObservable. Определяет коллекцию объектов наблюдателей.
* IObserver: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет метод Update(), который вызывается наблюдаемым объектом для уведомления наблюдателя.
* ConcreteObserver: конкретная реализация интерфейса IObserver.

**MEDIATOR (ПОСРЕДНИК)**

abstract class Mediator

{ public abstract void Send(string msg, Colleague colleague); }

abstract class Colleague

{

    protected Mediator mediator;

    public Colleague(Mediator mediator)

    { this.mediator = mediator    }

}

class ConcreteColleague1 : Colleague

{

    public ConcreteColleague1(Mediator mediator) : base(mediator)  { }

    public void Send(string message)

    { mediator.Send(message, this);    }

    public void Notify(string message) { }

}

class ConcreteColleague2 : Colleague

{

    public ConcreteColleague2(Mediator mediator) : base(mediator)  { }

    public void Send(string message)

    { mediator.Send(message, this);    }

    public void Notify(string message) { }

}

class ConcreteMediator : Mediator

{

    public ConcreteColleague1 Colleague1 { get; set; }

    public ConcreteColleague2 Colleague2 { get; set; }

    public override void Send(string msg, Colleague colleague)

    {

        if (Colleague1 == colleague)

            Colleague2.Notify(msg);

        else

            Colleague1.Notify(msg);

    }

}

* Mediator: представляет интерфейс для взаимодействия с объектами Colleague
* Colleague: представляет интерфейс для взаимодействия с объектом Mediator
* ConcreteColleague1 и ConcreteColleague2: конкретные классы коллег, которые обмениваются друг с другом через объект Mediator
* ConcreteMediator: конкретный посредник, реализующий интерфейс типа Mediator

**MEMENTO (ХРАНИТЕЛЬ)**

class Memento

{

    public string State { get; private set;}

    public Memento(string state)

    {

        this.State = state;

    }

}

class Caretaker

{

    public Memento Memento { get; set; }

}

class Originator

{

    public string State { get; set; }

    public void SetMemento(Memento memento)

    {

        State = memento.State;

    }

    public Memento CreateMemento()

    {

        return new Memento(State);

    }

}

* Memento: хранитель, который сохраняет состояние объекта Originator и предоставляет полный доступ только этому объекту Originator
* Originator: создает объект хранителя для сохранения своего состояния
* Caretaker: выполняет только функцию хранения объекта Memento, в то же время у него нет полного доступа к хранителю и никаких других операций над хранителем, кроме собственно сохранения, он не производит

**VISITOR (ПОСЕТИТЕЛЬ)**

abstract class Visitor {

    public abstract void VisitElementA(ElementA elemA);

    public abstract void VisitElementB(ElementB elemB); }

class ConcreteVisitor1 : Visitor {

public override void VisitElementA(ElementA elementA) { elementA.OperationA();   }

    public override void VisitElementB(ElementB elementB) { elementB.OperationB();   }

}

class ConcreteVisitor2 : Visitor

{

    public override void VisitElementA(ElementA elementA) { elementA.OperationA();   }

    public override void VisitElementB(ElementB elementB) { elementB.OperationB();   }

}

class ObjectStructure

{

    List<Element> elements = new List<Element>();

    public void Add(Element element) { elements.Add(element); }

    public void Remove(Element element) { elements.Remove(element);  }

    public void Accept(Visitor visitor)

    {

        foreach (Element element in elements)

            element.Accept(visitor);

    }

}

abstract class Element {

    public abstract void Accept(Visitor visitor);

    public string SomeState { get; set; } }

class ElementA : Element{

    public override void Accept(Visitor visitor) { visitor.VisitElementA(this);   }

    public void OperationA(){ } }

class ElementB : Element {

public override void Accept(Visitor visitor) { visitor.VisitElementB(this);    }

    public void OperationB() { }

}

* **Visitor**: интерфейс посетителя, который определяет метод Visit() для каждого объекта Element
* **ConcreteVisitor1 / ConcreteVisitor2**: конкретные классы посетителей, реализуют интерфейс, определенный в Visitor.
* **Element**: определяет метод Accept(), в котором в качестве параметра принимается объект Visitor
* **ElementA / ElementB**: конкретные элементы, которые реализуют метод Accept()
* **ObjectStructure**: некоторая структура, которая хранит объекты Element и предоставляет к ним доступ. Это могут быть и простые списки, и сложные составные структуры в виде деревьев

**STRATEGY (СТРАТЕГИЯ)**

public interface IStrategy

{

    void Algorithm();

}

public class ConcreteStrategy1 : IStrategy

{

    public void Algorithm()

    {}

}

public class ConcreteStrategy2 : IStrategy

{

    public void Algorithm()

    {}

}

public class Context

{

    public IStrategy ContextStrategy { get; set; }

    public Context(IStrategy \_strategy)

    {

        ContextStrategy = \_strategy;

    }

    public void ExecuteAlgorithm()

    {

        ContextStrategy.Algorithm();

    }

}

* Интерфейс **IStrategy,** который определяет метод Algorithm(). Это общий интерфейс для всех реализующих его алгоритмов. Вместо интерфейса здесь также можно было бы использовать абстрактный класс.
* Классы **ConcreteStrategy1** и **ConcreteStrategy**, которые реализуют интерфейс IStrategy, предоставляя свою версию метода Algorithm(). Подобных классов-реализаций может быть множество.
* Класс **Context** хранит ссылку на объект IStrategy и связан с интерфейсом IStrategy отношением агрегации.

**№6-7**

1. Перечислите преимущества и недостатки WPF?

►Веб-подобная модель компоновки

► Богатая модель рисования

► Развитая текстовая модель

► Поддержка аудио и видео

► Приложения на основе страниц

► Декларативный пользовательский интерфейс (XAML) (сочетание)

► Стили и шаблоны. Команды

1. Зачем нужен язык XAML? Каким образом он используется в WPF?

► eXtensible Application Markup Language (XAML)

Декларативный язык разметки для пользовательского интерфейса

Совместная работа дизайнера и разработчика с помощью инструментов

WCF , WF

►Каждый элемент в документе XAML отображается на экземпляр класса .NET.

►код XAML допускает вложение одного элемента внутрь другого ►Свойства каждого класса можно устанавливать через атрибуты или вложенные дискрипторы

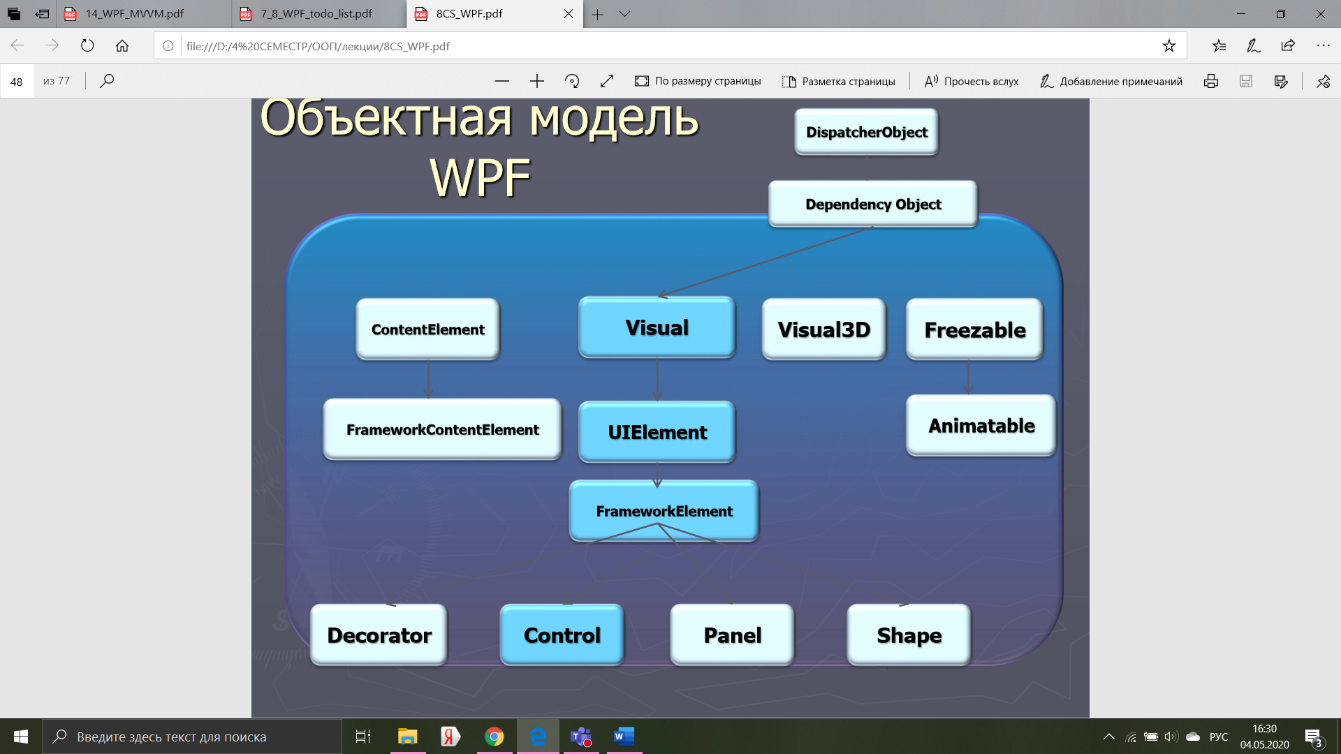
1. Какие бывают контейнеры компоновки?

**Grid**, **UniformGrid**, **StackPanel**, **WrapPanel**, **DockPanel** и **Canvas**. ViewBox

4. Перечислите основные группы элементов управления. Расскажите про объектную модель WPF.

Все элементы управления могут быть условно разделены на несколько подгрупп:

* **Элементы управления содержимым**, например кнопки (Button), метки (Label)
* **Специальные контейнеры**, которые содержат другие элементы, но в отличие от элементов Grid или Canvas не являются контейнерами компоновки - ScrollViewer,GroupBox
* **Декораторы**, чье предназначение создание определенного фона вокруг вложенных элементов, например, Border или Viewbox.
* **Элементы управления списками**, например, ListBox, ComboBox.
* **Текстовые элементы управления**, например, TextBox, RichTextBox.
* **Элементы, основанные на диапазонах значений**, например, ProgressBar, Slider.
* **Элементы для работ с датами**, например, DatePicker и Calendar.
* **Остальные элементы управления**, которые не вошли в предыдущие подгруппы, например, Image.



* 1. Объясните назначение класса DependencyProperty. Зачем нужны свойства зависимостей? Как создать новой свойство зависимости и в каких случаях это необходимо?

Без свойств зависимостей были бы невозможны многие ключевые особенности WPF, как привязка данных, стили, анимация и т.д.

Сначала нужно определить объект, который будет представлять свойство. Это экземпляр класса DependencyProperty.

Чтобы его можно было задействовать, необходимо зарегистрировать свойство зависимости в WPF.

Регистрация свойства зависимости осуществляется в два этапа. Сначала создается объект FrameworkPropertyMetadata, который указывает, какие службы вы хотите использовать со свойством зависимости (например, поддержку привязки данных, анимацию и ведение журнала). Затем свойство регистрируется, для чего вызывается метод DependencyProperty.Register()

* 1. Каким образом осуществляется обработка событий в WPF? Что означает концепция маршрутизированных событий? Перечислите основные группы событий.

События, возникнув на одном элементе, могут обрабатываться на другом. События могут подниматься и опускаться по дереву элементов.

Так, маршрутизируемые события делятся на три вида:

* **Прямые** (direct events) - они возникают и отрабытывают на одном элементе и никуда дальше не передаются. Действуют как обычные события.
* **Поднимающиеся** (bubbling events) - возникают на одном элементе, а потом передаются дальше к родителю - элементу-контейнеру и далее, пока не достигнет наивысшего родителя в дереве элементов.
* **Опускающиеся, туннельные** (tunneling events) - начинает отрабатывать в корневом элементе окна приложения и идет далее по вложенным элементам, пока не достигнет элемента, вызвавшего это событие.

Все маршрутизируемые события используют класс **RoutedEventArgs** (или его наследников),

* 1. Что такое Resource Dictionary?

Чтобы разделить ресурсы между множеством проектов можно создать словарь ресурсов. ***Словарь ресурсов*** представляет собой просто XAML-документ, который всего лишь хранит необходимые ресурсы.

* 1. Что такое стиль и как его создать? В чем преимущество использования стиля?

Стили позволяют определить набор некоторых свойств и их значений, которые потом могут применяться к элементам в xaml.

 Стили помогают создать стилевое единообразие для определенных элементов.

Стиль создается как ресурс с помощью объекта **Style.**

* 1. Что такое Command? Расскажите паттерн Command. Как в WPF используется Command и для чего?

Паттерн "Команда" (Command) позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта.

в технологии WPF и других технологиях, которые используют XAML и подход MVVM, на командах во многом базируется взаимодействие с пользователем.

Использование команд помогает нам сократить объем кода и использовать одну и ту же команду для нескольких элементов управления в различных местах программы. Таким образом, команды позволяют абстрагировать набор действий от конкретных событий конкретных элементов.

**№8**

1. Для чего в WPF используются ресурсы? Каким образом можно определить новый ресурс и управлять им? Опишите назначение класса ResourceDictionary. Зачем каждый элемент имеет собственную коллекцию ресурсов?
2. ***Система ресурсов*** WPF представляет собой простой способ поддержания вместе набора полезных объектов, таких как наиболее часто используемые кисти, стили или шаблоны, что существенно упрощает работу с ними.
3. В WPF ресурсы можно определять в коде или в различных местах внутри разметки (вместе с отдельными элементами управления, внутри отдельных окон или во всем приложении).

Каждый элемент включает свойство **Resources**, в котором хранится словарная коллекция ресурсов (представляющая собой экземпляр класса ResourceDictionary). Эта коллекция ресурсов может хранить объект любого типа с индексацией по строке.

2) Какая разница между статическими и динамическими ресурсами?

Статические ресурсы устанавливается только один раз. А динамические ресурсы могут меняться в течение работы программы.

3) Что такое триггеры ? Для чего в WPF используются триггеры? Назовите основные типы триггеров.

* Триггеры – декларативное определение некоторых действий, которые выполняются при изменении свойств (свойств зависимостей) стиля
* **Триггеры свойств**: вызываются в ответ на изменения свойствами зависимостей своего значения
* **Триггеры данных**: вызываются в ответ на изменения значений любых свойств (они необязательно должны быть свойствами зависимостей)
* **Триггеры событий**: вызываются в ответ на генерацию событий
* **Мультитриггеры**: вызываются при выполнении ряда условий

1. Что такое локализация и как ее обеспечить.

Локализация — это перевод ресурсов приложения в локализованные версии для конкретных языков и региональных параметров, которые поддерживает приложение. При локализации в WPFиспользуются API-интерфейсы в пространстве имен [System.Windows.Markup.Localizer](https://docs.microsoft.com/ru-ru/dotnet/api/system.windows.markup.localizer). Эти API-интерфейсы.

1. Что такое тема? Опишите процесс создания темы на основе ресурсов и стилей.

Темы ►объединение стилей

1. Создается файл словаря ресурсов nigth . xaml , и определяется некоторый набор ресурсов:
2. ►2) Выводим ЭУ для смены тем и меняем

6) Что такое шаблон и как его создать?

Шаблон - средство, определяющее способ визуализации объекта.

Все визуальные элементы в WPF уже имеют встроенные шаблоны, которые определяют визуальное дерево, структуру и даже поведение элементов

Мы можем определять шаблоны с через стили, а можем в виде отдельных ресурсов.

7)Зачем нужны пользовательские элементы управления? Как создать собственный элемент. Опишите члены класса UserControl.

Пользовательские элементы управления являются основной единицей повторно используемого Xaml и прилагающегося к нему кода.

8) Что такое привязка данных? В чем разница между следующими режимами привязки: OneWay, TwoWay и OneTime.

Привязка подразумевает взаимодействие двух объектов: источника и приемника. Объект-приемник создает привязку к определенному свойству объекта-источника. В случае модификации объекта-источника, объект-приемник также будет модифицирован.

Свойство **Mode** объекта Binding, которое представляет режим привязки, может принимать следующие значения:

* **OneWay**: свойство объекта-приемника изменяется после модификации свойства объекта-источника.
* **OneTime**: свойство объекта-приемника устанавливается по свойству объекта-источника только один раз. В дальнейшем изменения в источнике никак не влияют на объект-приемник.
* **TwoWay**: оба объекта - применки и источник могут изменять привязанные свойства друг друга.

9) Объясните назначение интерфейса INotifyPropertyChanged

 Чтобы объект мог полноценно реализовать механизм привязки, нам надо реализовать в его классе интерфейс INotifyPropertyChanged.

Когда объект класса изменяет значение свойства, то он через событие PropertyChanged извещает систему об изменении свойства.

**№9**

1. Что такое Свойства зависимости? Для чего они нужны?

Свойства зависимостей предназначены для предоставления способа вычисления значения свойства по значениям других входных данных.

Эти входные данные могут включать в себя системные свойства, такие как темы и пользовательские параметры, JIT-механизмы определения свойств, такие как привязка данных и анимации (раскадровки), шаблоны многократного использования, например ресурсы и стили, а также значения, известные благодаря отношениям между родительскими и дочерними элементами с другими элементами в дереве.

1. Как создать DependencyProperty?

Если мы хотим применять свойства зависимостей, то нам надо унаследовать свой класс от абстрактного класса DependencyObject. В нашем классе мы определяем два свойства зависимостей: TitleProperty и PriceProperty. Обратите внимание, что они объявляются с модификаторами public static readonly.

Затем свойства регистрируются в статическом конструкторе нашего класса с помощью метода Register. И в конце для них создаются обычные свойства-обертки, в которых мы получаем доступ к значению свойств с помощью методов GetValue и SetValue.

1. Для чего и как используют делегат ValidateValueCallback?

Делегат, который возвращает true, если значение проходит валидацию, и false - если не проходит.

1. Для чего и как используют делегат CoerceValueCallback?

Делегат, который может подкорректировать уже существующее значение свойства, если оно вдруг не попадает в диапазон допустимых значений.

1. Какие типы маршрутизируемых событий есть в WPF (поясните каждый)?

* Прямые (direct events) - они возникают и отрабытывают на одном элементе и никуда дальше не передаются. Действуют как обычные события.
* Поднимающиеся (bubbling events) - возникают на одном элементе, а потом передаются дальше к родителю - элементу-контейнеру и далее, пока не достигнет наивысшего родителя в дереве элементов.
* Опускающиеся, туннельные (tunneling events) - начинает отрабатывать в корневом элементе окна приложения и идет далее по вложенным элементам, пока не достигнет элемента, вызвавшего это событие.

1. Как создать RoutedEvent?

Для определения маршрутизированных событий в классе создвалось статическое поле по типу RoutedEvent.

Это поле, как правило, имеет суффикс Event. Затем это событие регистрируется в статическом конструкторе.

И также класс, в котором создается событие, как правило определяет объект-обертку над обычным событием. В этой обертке с помощью метода AddHandler происходит добавление обработчика для данного события, а с помощью метода RemoveHandler - удаление обработчика.

1. Поясните концепцию Command в WPF? В чем ее преимущества?

Команды представляют механизм выполнения какой-нибудь задачи. Использование команд помогает нам сократить объем кода и использовать одну и ту же команду для нескольких элементов управления в различных местах программы. Таким образом, команды позволяют абстрагировать набор действий от конкретных событий конкретных элементов.

1. Как используются команды?

См. вопрос 7.

1. Как создать RoutedUICommand?

WPF уже обладает большим набором встроенных команд. Все они представляют объекты класса RoutedUICommand, который является производным от RoutedCommand.

**№10**

1. Что такое Binding в WPF? Какие атрибуты мы можем указывать и для чего?

Привязка подразумевает взаимодействие двух объектов: источника и приемника. Объект-приемник создает привязку к определенному свойству объекта-источника.

Некоторые свойства класса **Binding**:

* **ElementName**: имя элемента, к которому создается привязка
* **IsAsync**: если установлено в True, то использует асинхронный режим получения данных из объекта. По умолчанию равно False
* **Mode**: режим привязки
* **Path**: ссылка на свойство объекта, к которому идет привязка
* **TargetNullValue**: устанавливает значение по умолчанию, если привязанное свойство источника привязки имеет значение null
* **RelativeSource**: создает привязку относительно текущего объекта
* **Source**: указывает на объект-источник, если он не является элементом управления.
* **XPath**: используется вместо свойства path для указания пути к xml-данным

1. Что такое ADO.NET? Какие существуют режимы работы?

ADO.NET предоставляет собой технологию работы с данными, которая основана на платформе .NET Framework. Эта технология представляет нам набор классов, через которые мы можем отправлять запросы к базам данных, устанавливать подключения, получать ответ от базы данных и производить ряд других операций.

Функционально классы ADO.NET можно разбить на два уровня: подключенный и отключенный.

Подключенный

►Работает в режиме удержания подключения к базе.

►Обеспечивает максимальную гибкость и эффективность.

►Обеспечивает минимальный расход оперативной памяти.

Отключенный

►Обеспечивает работу с данными в отсутствии подключения к БД.

►Удобна для переноса данных по сети.

►Расходует достаточно много памяти

1. Объясните назначение класса SqlConnection.

Обеспечивают подключение к БД.

1. Объясните назначение класса SqlDataAdapter

Заполняет отсоединенный объект DataSet или обновляют данные на сервере из DataSet.

1. Объясните назначение класса SqlException. Чем данный класс отключается от класса-родителя Exception?

Исключение, которое возникает, когда SQL Server возвращает предупреждение или ошибку. Этот класс не может быть унаследован.

1. Объясните назначение класса SqlCommand.

Управляет источником данных с помощью SQL.

1. В чем разница между методами ExecuteNonQuery, ExecuteReader и ExecuteScalar?

* ExecuteScalar обычно используется, когда ваш запрос возвращает одно значение. Если он возвращает больше, то результатом будет первый столбец первой строки. Примером может быть SELECT @@IDENTITY AS 'Identity'.
* ExecuteReader используется для любого набора результатов с несколькими строками/столбцами (например, SELECT col1, col2 from sometable).
* ExecuteNonQuery обычно используется для операторов SQL без результатов (например, UPDATE, INSERT и т.д.).

1. Объясните назначение класса SqlDataReader.

Обеспечивает последовательный доступ к результату выполнения команды Select. Для работы требуют монопольный доступ к соединению.

1. Объясните назначение класса SqlParameter.

Представляет параметр для [SqlCommand](https://docs.microsoft.com/en-us/dotnet/api/system.data.sqlclient.sqlcommand?view=dotnet-plat-ext-3.1) и, при необходимости, его сопоставление со столбцами [DataSet](https://docs.microsoft.com/en-us/dotnet/api/system.data.dataset?view=dotnet-plat-ext-3.1) .

1. Объясните назначение класса SqlTransaction.

Представляет транзакцию Transact-SQL, которая должна быть сделана в базе данных SQL Server.

1. Объясните назначение классов DataSet, DataTable, DataColumn, DataRow, DataRelation.

Объекты DataTable используются для представления одной из таблиц базы данных в DataSet.

В свою очередь, DataTable составляется из объектов DataColumn. DataColumn - это блок для создания схемы DataTable.

Объекты DataRelation

Объект DataSet имеет также свойство Relations, возвращающее коллекцию DataRelationCollection, которая в свою очередь состоит из объектов DataRelation. Каждый объект DataRelation выражает отношение между двумя таблицами (сами таблицы связаны по какому-либо полю (столбцу)). Следовательно, эта связь осуществляется через объект DataColumn.

Строки (объект DataRow)

Коллекция Rows объекта DataTable возвращает набор строк (записей) заданной таблицы.

**№12**

* 1. Для чего используют многоуровневые архитектуры?

При этом надо отметить, что крайние уровни не могут взаимодействовать между собой, то есть уровень представления (применительно к ASP.NET MVC, контроллеры) не могут напрямую обращаться к базе данных и даже к уровню доступа к данным, а только через уровень бизнес-логики.

* 1. Опишите назначение слоев многоуровневой архитектуре. Business layer (уровень бизнес-логики), Data Access layer (уровень доступа к данным).

**Business layer** (уровень бизнес-логики): содержит набор компонентов, которые отвечают за обработку полученных от уровня представлений данных, реализует всю необходимую логику приложения, все вычисления, взаимодействует с базой данных и передает уровню представления результат обработки.

**Data Access layer** (уровень доступа к данным): хранит модели, описывающие используемые сущности, также здесь размещаются специфичные классы для работы с разными технологиями доступа к данным, например, класс контекста данных Entity Framework. Здесь также хранятся репозитории, через которые уровень бизнес-логики взаимодействует с базой данных.

* 1. Поясните назначение и принцип использования паттерна Repository?

 Репозиторий позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой.

* 1. Поясните назначение и принцип использования паттерна Unit of Work?

Паттерн Unit of Work позволяет упростить работу с различными репозиториями и дает уверенность, что все репозитории будут использовать один и тот же контекст данных.

* 1. Опишите основные особенности библиотеки Entity Framework?

Entity Framework представляет ORM-технологию, которая позволяет абстрагироваться от структуры базы данных и может выполнять автоматически сопоставление таблиц и их данных с моделями классов, определенных разработчиком.

* 1. Какие преимущества обеспечивает использование слабосвязанного кода?

 Слабо связанный программный код – это код, в котором объекты довольно независимы друг от друга. Если объект сильно связан с другими, то во-первых его приходится часто модифицировать, так как малейшие изменения окружения требуют этого, а во-вторых затрудняется его повторное использование, поскольку при повторном использовании надо восстановить все его связи. В сильно связанном программном коде затруднительно рассмотрение каждого объекта в отдельности.

Как правило, репозиторий привязан к одной конкретной сущности или модели, данными которой он управляет. Хотя это необязательно - в репозитории мы можем предусмотреть механизм для загрузки связанных данных из других таблиц, которые связаны с основной моделью, и ряд аналогичных операций. Но тем не менее, часто для управлению одной сущность создается свой репозиторий.

Если репозитории используют одно и то же подключение, то нередко для организации доступа к одному подключению для всех репозиториев приложения используется другой паттерн - **Unit Of Work**. Класс, который реализует данный паттерн, как правило, содержит набор репозиториев и ряд некоторых общих для них функций.

Юзер.сс

Но если мы обратимся непосредственно к Entity Framework, то мы можем увидеть, то он уже реализует паттерны Unit Of Work и репозиторий. К примеру, контекст данных EF для выше обозначенных моделей мог бы выглядеть следующим образом.

Майдбсонтекст

Фактически класс майдбContext представляет реализацию UnitOfWork - он содержит ряд репозиториев. Каждый репозиторий представлен объектом DbSet, с помощью функциональности которого мы можем получать, добавлять, удалять данные.

К примеру реализуем паттерн репозиторий для работы через EF 6.

контроллеры

**№14**

1. Что такое UML?

**UML** – унифицированный язык моделирования (Unified Modeling Language) – это система обозначений, которую можно применять для объектно-ориентированного анализа и проектирования.

1. Перечислите типы диаграмм UML.

* **Диаграмма** вариантов использования (use case **diagram**)
* **Диаграмма** классов (class **diagram**)
* **Диаграмма** состояний (statechart **diagram**)
* **Диаграмма** последовательности (sequence **diagram**)
* **Диаграмма** кооперации (collaboration **diagram**)
* **Диаграмма** компонентов (component **diagram**)
* **Диаграмма** развертывания (deployment **diagram**)

1. Какие отношения между классами могут быть на диаграмме классов?

Наследование, агрегация, композиция, ассоциация, реализация.

Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского).

в незакрашенной стрелочке

Реализация предполагает определение интерфейса и его реализация в классах.

в незакрашенной стрелочке от класса к интерфейсу, только линия теперь пунктирная

1. Как обозначаются абстрактные классы на диаграмме классов?

 Если **класс абстрактный** — то его имя пишется полужирным курсивом.

5. Как обозначаются интерфейсы на диаграмме классов?

См записку

6.Как отображается доступность членов класса на диаграмме классов?

Так, например, на рисунке ниже атрибут “Атрибут1” является открытым (+) членом класса, а атрибут “Атрибут2” – закрытым (-).

7. Что такое агрегация? Как обозначается?

Композиция определяет отношение HAS A, то есть отношение "имеет". Например, в класс автомобиля содержит объект класса электрического двигателя.

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны.

 ромбик будет незакрашенным

8. Что такое ассоциация?

Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа.

На схемах UML ассоциация обозначается в виде обычно стрелки.

1. Какие обозначения используют на диаграмме последовательности?

Основными элементами диаграммы последовательности являются обозначения [объектов](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (прямоугольники с названиями объектов), вертикальные «линии жизни» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *lifeline*), отображающие течение времени, прямоугольники, отражающие деятельность объекта или исполнение им определенной функции (прямоугольники на пунктирной «линии жизни»), и стрелки, показывающие обмен сигналами или сообщениями между объектами.

1. Для чего нужна диаграмма последовательности?

**Диаграмма последовательности** действий (sequence diagram) отображает взаимодействие объектов, упорядоченное по времени.

1. Каково назначение диаграммы использования, пакетов и активности?

*диаграммы пакетов* (package diagrams) служат для разбиения объемных моделей на составные части, а также (традиционно) для группировки классов моделируемого ПО, когда их слишком много.

*диаграммы активностей* (*activity diagrams*) используются для спецификации бизнес-процессов, которые должно автоматизировать разрабатываемое ПО, а также для задания сложных алгоритмов;

Диаграмма вариантов использования в UML — диаграмма, отражающая отношения между актерами и прецедентами и являющаяся составной частью модели прецедентов, позволяющей описать систему на концептуальном уровне.