**№01. Основы ООП. Классы и объекты. Синтаксис описания классов.**

**ООП** – парадигма программирования, в которой основными концепциями являются понятия объекта и класса. Механизмы ООП:

* *Абстракция*. Позволяет получить упрощённую модель объекта реального мира, путем выделения основных важных свойств и поведения, и отделения всего остального. В ЯП абстракция реализована в механизме создания классов и интерфейсов.
* *Наследование*. Позволяет создавать иерархии в модели. При наследовании все поведение базового класса заимствуется и включается в наследник.
* *Полиморфизм*. В общем смысле этот принцип можно описать как «один интерфейс – множество форм». Полиморфизм помогает упростить программу, позволяя использовать один и тот же интерфейс для группы взаимосвязанных действий. Таким образом, при создании иерархии мы можем иметь несколько классов с одинаковыми методами и различной их реализацией. В ЯП полиморфизм реализован через определение виртуальных функций.
* *Инкапсуляция*. Суть его в том, чтобы скрыть доступ к определенным элементам нашей программы – свойствам, полям или методам. В ЯП инкапсуляция реализована при помощи модификаторов доступа – public, private, protected и т.п.

**Класс** – пользовательский тип данных, который состоит из данных полей и членов, оперирующими этими данными (конструкторов, свойств, методов, событий и т.д.).

**Объект** – это экземпляр класса, который обладает неким состоянием и поведением.

Простейшее описание класса в языке C# выглядит так:

class NewClass

{

}

Более сложный класс в языке C# может содержать:

* *до ключевого слова class*: атрибуты и модификаторы класса.
* *после имени класса*: параметры обобщённого типа, базовый класс и интерфейсы.
* *внутри фигурных скобок*: члены класса (поля, свойства, индексаторы, методы, события и т.п.).

**№02. ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ В ООП. ПОЛИМОРФИЗМ.**

Можно выделить следующие отношения между классами:

* *Наследование (отношение типа «является»)*. Объекты дочернего класса наследуют все свойства родительского класса.
* *Реализация интерфейса (отношение типа «поддерживает»)*.
* *Агрегация (отношение типа «имеет»)*. Отображает отношение между целым и его частями, т.е. объекты одного класса входят в объекты другого.
* *Композиция*. Более строгий вариант агрегации: объекты одного класса входят в объекты другого и зависят друг от друга по времени жизни.
* *Ассоциация*. Объекты классов вступают во взаимодействие между собой. Пример: руководитель – студент. Множественность: 1-n, 1-1, n-m.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

В общем смысле этот принцип можно описать как «один интерфейс – множество форм». Полиморфизм помогает упростить программу, позволяя использовать один и тот же интерфейс для группы взаимосвязанных действий. Таким образом, при создании иерархии мы можем иметь несколько классов с одинаковыми методами и различной их реализацией.

Поддержка полиморфизма в C#:

* Переопределение метода (ключевые слова **virtual** (БК – разрешение переопределения), **override** (ДК – намерение переопределить) и **sealed)**;
* Абстрактные классы (**abstract**). Объект абстрактного класса создать нельзя. Абстрактные классы могут иметь абстрактные члены (в отличии от виртуальных не имеют реализации по умолчанию);

**№03. Интерфейсы как элемент ООП.**

Рассмотрим концепцию интерфейсов на примере языка C#. Объявление интерфейса аналогично объявлению класса, но оно не содержит реализации членов. Эти члены будут реализованы классами и структурами, реализующими интерфейс.

public interface IClonable

{

object Clone();

}

class Employee : Person, IClonable

{

public object Clone()

{

//to do something;

}

}

Интерфейс аналогичен классу, но он предоставляет только спецификацию (а не реализацию) своих членов.

Особенности:

* класс может реализовывать несколько интерфейсов. Однако наследоваться он может только от одного класса;
* все члены интерфейса неявно абстрактные. В противоположность этому, члены класса могут быть как абстрактными, так и содержать конкретную реализацию.
* структуры могут реализовывать интерфейс, однако не могут наследоваться от класса.
* интерфейс может содержать только те члены, которые в классе могут быть помечены как abstract.
* члены интерфейсов не могут иметь модификаторов доступа.
* интерфейсы могут быть произведены от других интерфейсов.

**№04. СОБЫТИЙНОЕ ПРОГРАММИРОВАНИЕ. ОПИСАНИЕ И ИСПОЛЬЗОВАНИЕ СОБЫТИЙ**

Рассмотрим события на примере языка C#.

**Делегат** – пользовательский тип данных, который инкапсулирует метод. В C# объявление типа делегата начинается с ключевого слова delegate, а в остальном напоминает объявление абстрактного метода: delegate int Transformer(int x);

**События** – способ описания связи одного объекта с другими по действиям. Событие можно объявить в пределах класса, структуры или интерфейса. Базовый синтаксис объявления события следующий:

*<модификаторы>* ***event*** *<тип делегата> <имя события>;*

Ключевое слово **event** указывает на объявление события. При объявлении события требуется указать делегат, описывающий метод обработки события. Этот делегат должен иметь тип возвращаемого значения **void**.

Для генерации события в требуемом месте кода помещается вызов в формате *<имя события>(<фактические аргументы>)*. Предварительно обычно проверяют, назначен ли обработчик события. Генерация события может происходить только в том же классе, в котором событие объявлено.

Регистрация получателя события. Чтобы отреагировать на событие, его надо ассоциировать с обработчиком события. Назначение и удаление обработчиков события выполняется при помощи операторов += и -=.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Платформа .NET определяет стандартный способ написания событий. В центре находится класс **System.EventArgs**, который является базовым для передачи инф-ции событию. . В частности, для типов событий зарезервированы следующие делегаты:

public delegate void EventHandler(object sender, EventArgs e);

public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e) where TEventArgs : EventArgs;

Как видим, данные делегаты предполагают, что **первым** параметром будет выступать **объект, в котором событие было сгенерировано**. **Второй** параметр используется для **передачи информации события**. Это либо класс EventArgs, либо наследник этого класса с необходимыми полями.

Сама генерация события обычно выносится в отдельный защищенный виртуальный метод класса. В этом методе проверяется, был ли установлен обработчик события.

**№05. ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ. ПРИМЕРЫ ГЕНЕРАЦИИ И ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ.**

Для генерации исключительной ситуаций в языке C# используется оператор **throw**, после которого указывается объект класса исключительной ситуации. В C# классами исключительных ситуаций являются класс **System.Exception** и все его наследники. Класс **Exception** – это базовый класс для представления исключительных ситуаций.

В пространстве имён System содержится несколько классов для описания наиболее распространённых исключений:

* **ArgumentException** генерируется, когда методу передаётся недопустимый аргумент.
* **ArgumentNullException** (наследник ArgumentException) генерируется, когда методу передаётся аргумент, равный null.
* **ArgumentOutOfRangeException** (наследник ArgumentException) генерируется, когда методу передаётся аргумент, выходящий за допустимый диапазон.
* **InvalidOperationException** сигнализирует о том, что состояние объекта препятствует выполнению метода (пример: запись в файл, который открыт только для чтения).
* **NotImplementedException** сигнализирует о том, что функциональная возможность не реализована.

Разработчик может создать собственный класс для представления информации об исключительной ситуации. Единственным условием для этого класса в C# является прямое или косвенное наследование от класса Exception.

Опишем возможности по обработке исключительных ситуаций. Для перехвата исключительных ситуаций служит блок try – catch – finally. Синтаксис блока следующий:

try

{

… //тут может быть возбуждено исключение

}

catch (ExceptionA ex)

{

… //обработка исключения ExceptionA

}

finally

{

… //завершающий код

}

Если один из операторов, расположенных в блоке try, вызвал исключительную ситуацию, управление немедленно передается на блоки catch. Здесь ex – это некая временная переменная, которая может использоваться для извлечения информации из объекта исключительной ситуации.

Если указать в блоке catch оператор throw, это приведёт к тому, что обрабатываемая исключительная ситуация будет сгенерирована повторно.

**Оператор using.** Многие классы инкапсулируют неуправляемые ресурсы (например, соеденения с БД). Эти классы реализуют интерфейс System.IDisposeable, который определяет единственный метод Dispose, предназначенный для корректного завершения работы с этими ресурсами. Оператор using предоставляет элегантный синтаксис создания объекта IDisposeable с последующим выводом метода Dispose.

using (StreamReader reader = File.OpenTxt(“file.txt”))

{

…

}

Эквивалентно:

StreamReader reader = File.OpenTxt(“file.txt”));

try { … }

finally { if (reader != null) ((IDisposable)reader).Dispose();}

**№06. ОБЩАЯ ХАРАКТЕРИСТИКА ПЛАТФОРМЫ .NET**

Платформа .NET образует каркас, включающий библиотеку классов, технологии для доступа к данным и построения оконных и веб-приложений. Основным инструментом разработки для платформы .NET является интегрированная среда Microsoft Visual Studio.

База платформы .NET - это **общеязыковая среда исполнения** (Common Language Runtime, CLR). CLR является «прослойкой» между операционной системой и приложением.

Приложения для платформы .NET состоят из **управляемого кода** (managed code). Управляемый код является результатом компиляции исходных текстов. Скомпилированные файлы называются **сборками** (assembly) и включают следующие части:

1. *Манифест* (manifest) – описание сборки: версия, ограничения безопасности, список необходимых внешних сборок.
2. *Метаданные* – специальное описание всех пользовательских типов, размещённых в сборке.
3. *Код на языке Microsoft Intermediate Language* (MSIL, или просто IL). Данный код является независимым от операционной системы и типа процессора. В процессе работы приложения он компилируется в машинно-зависимый код специальным JIT-компилятором (Just-in-Time compiler).

Основная задача CLR – это манипулирование сборками: загрузка, JIT-компиляция, создание окружения для выполнения сборок. Важной функцией CLR является **управление памятью** при работе приложения и выполнение **автоматической сборки мусора**, то есть фонового освобождения неиспользуемой памяти. Кроме этого, CLR реализует в приложениях для .NET проверку типов, управление политиками безопасности при доступе к коду и другие функции.

В состав платформы .NET входит **обширная библиотека классов** Framework Class Library (FCL). Элементом этой библиотеки является **базовый набор классов** Base Class Library (BCL). В BCL входят классы для работы со строками и коллекциями данных, для поддержки многопоточности и множество других классов. Частью FCL являются компоненты, поддерживающие различные технологии обработки данных и организации взаимодействия с пользователем. Это классы для работы с XML и базами данных, для создания пользовательских интерфейсов.

В стандартную поставку платформы .NET включено **несколько компиляторов**. Это компиляторы языков C#, F#, Visual Basic .NET, C++/CLI.

Для поддержки межъязыкового взаимодействия служат две спецификации платформы .NET.

* *Общая система типов* (Common Type System, CTS) описывает набор типов, который должен поддерживаться любым языком программирования для .NET.
* *Общеязыковая спецификация* (Common Language Specification, CLS) – это общие правила поведения для всех .NET-языков.

**№07. ОСНОВНЫЕ СИНТАКСИЧЕСКИЕ КОНСТРУКЦИИ ЯЗЫКА C#.**

Синтаксис языка C# основан на синтаксисе таких C-подобных языков. C# является регистрозависимым языком.

* **Идентификаторы**. Слово из символов UNICODE, начинающееся с буквы или символа подчеркивания.
* **Ключевые слова**. Зарезервированные компилятором имена. Их нельзя использовать в качестве идентификаторов (исключение @class, символ @ не считается частью идентификатора).
* **Контекстные ключевые слова**. Например, var.
* **Литералы** – простейшие элементы данных, статически встроенные в программу.
* **Пунктуаторы**. Помогают разметить программу.
* **Знаки операций.**
* **Комментарии.**
* **Операторы**
  + **Операторы объявления.** Объявляет новую переменную. Должен заканчиваться точкой с запятой.
  + **Операторы выражений:** выражения присваивания, выражения, вызывающие методы, выражения, создающие объекты.
  + **Операторы выбора:** if, switch
  + **Условная операция** (?:);
  + **Операторы цикла:** while, do-while, for, foreach.
  + **Операторы перехода:** break, continue, goto, return, throw.

**№08. КЛАССИФИКАЦИЯ ТИПОВ ПЛАТФОРМЫ .NET И ЯЗЫКА C#. СИНТАКСИС ОПИСАНИЯ ТИПОВ НА ЯЗЫКЕ C#.**

Все типы C# соответствуют определенным типам CLR.

С точки зрения размещения в памяти все типы делятся на:

* **Типы-значения** (числовые типы, bool, struct, enum). Переменная этого типа содержит данные и размещается в стеке. Когда один тип-значения присваивается другому, получается почисленная копия данных полей.
* **Ссылочные типы** (class, interface, string, array, delegate, object). Переменная этого типа содержит ссылку на данные, которые размещены у управляемой динамической памяти.

1. **Примитивные типы** (числовые, bool, string, object)
2. **Пользовательские типы** описываются с помощью особых синтаксических конструкций. Это класс, структура, интерфейс, массив, перечисление, делегат.

C# - язык со строгой типизацией. В случае несоответствия применяется операция преобразования типов.

**№09. УНИВЕРСАЛЬНЫЕ ШАБЛОНЫ В .NET И ЯЗЫКЕ C#.**

Generics позволяют конкретизировать тип при использовании, а не при разработке. Универсальные шаблоны нужны для создания кода, который можно многократно использовать с разными типами. Они могут быть применены к методам, делегатам, структурам, классам, интерфейсам.

Для описания универсального шаблона для класса используется следующий синтаксис: после имени класса в угловых скобках указывается параметр типа. Этот параметр может затем использоваться при описании элементов класса.

При объявлении универсального шаблона можно использовать несколько параметров-типов.

public class Dictionaty<K, V>

{

public void Add(K key, V value) { ... }

public V this[K key] { ... }

}

Язык С# допускает **указание ограничения** для каждого параметра универсального типа. Порядок элементов в списке ограничения имеет значение. Ограничения объявляются при помощи **ключевого слова where**, после которого указывается **параметр, двоеточие и список ограничений** (ключевое слово class (ссылочный), struct (тип-значение), имя класса, интерфейс или список интерфейсов, конструкция new() – чтобы у типа был конструктор без параметров).

Также есть возможность параметризировать не весь класс, а только отдельный метод.

**№10. ЖИЗНЕННЫЙ ЦИКЛ ОБЪЕКТОВ. АЛГОРИТМЫ «СБОРКИ МУСОРА».**

Переменные типов значений создаются в стеке, и их время жизни ограничено тем блоком кода, в котором они объявляются. Жизненный цикл объектов ссылочного типа начинается с ключевого слова new, провоцирующего неявный вызов специального метода класса - **конструктора**, который размещают объекты в динамической памяти и выполняют инициализацию полей объекта. Конструкторы не наследуются, и вызов конструктора в виде <имя объекта>.<имя конструктора> после создания объекта запрещен. Различают несколько видов конструкторов:

* **Конструктор по умолчанию** автоматически создаётся компилятором, если программист не описал в классе собственный конструктор. Конструктор по умолчанию – это всегда конструктор без параметров.
* **Пользовательский конструктор** описывается в классе как метод с именем, совпадающим с именем класса. У конструктора отсутствует любое указание на тип возвращаемого значения, даже **void**. Пользовательский конструктор может получать параметры, необходимые для инициализации объекта. Класс может содержать несколько пользовательских конструкторов, однако они обязаны различаться сигнатурой. Пользовательские конструкторы могут применяться для начальной инициализации **readonly-полей**. Если в классе определён хотя бы один пользовательский конструктор, **конструктор по умолчанию уже не создается**. Конструктор класса **может вызывать другой конструктор** того же класса, но только в начале своей работы. Для этого при описании конструктора используется синтаксис: public Pet() : this(10, "Pet") { . . . }
* **Статические конструкторы** используются для начальной инициализации статических полей класса. Статический конструктор объявляется с модификатором static и без параметров. Статические конструкторы **вызываются не программистом**, а CLR в следующих случаях:
  + перед созданием первого объекта класса или при первом обращении к элементу класса, не унаследованному от предка;
  + перед первым обращением к статическому полю, не унаследованному от предка.

В теле статического конструктора возможна **работа только со статическими полями и методами** класса; статические конструкторы **не могут вызывать экземплярные конструкторы** класса.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

После создания объект будет автоматически удален сборщиком мусора тогда, когда в нем отпадет необходимость. Это происходит, когда объект становится недостижимым ни в одной части программного кода. Однако нельзя с полной уверенностью полагать, что удаление произойдет немедленно. Все, что можно предположить, так это то, что когда в следующий раз, когда будет производится сборка мусора, данный объект может попасть под процесс уничтожения.

Для оптимизации поиска недостижимых объектов каждый объект в куче относят к некоторому «**поколению**». Чем дольше объект существует в куче, тем больше вероятность того, что он там останется.

* **Поколение 0**: Новый созданный объект, который никогда не помечается для сборки.
* **Поколение 1**: Объект, уже переживший сборку мусора (был помечен для сборки, но удален не был)
* **Поколение 2**: Объект, переживший более одной очистки памяти сборщиком мусора.

Сначала исследуется поколение 0, если в результате очистки этих объектов получено необходимое количество памяти, то все не удалённые объекты переходят в поколение 1. Если места все же не хватает, проверяется поколение 1. Пережившие сборку в поколении 1 объекты переходят в поколение 2. В последнюю очередь проверяется поколение 2. Объекты, пережившие сборку здесь, относятся к поколению 2, т.к. более высокие поколения просто не поддерживаются.

*Предназначение сборщика мусора* – управление памятью вместо нас. В языке C# есть возможность вызвать принудительную сборку мусора: **GC.Collect()** (в качестве параметра метод может принимать номер поколения, для которого следует произвести очистку).

В базовом классе System.Object имеется виртуальный метод **Finalize()**. Переопределение этого метода дает пользовательскую логику по очистке объекта. Метод Finalize() будет вызываться автоматически при сборке мусора. Для некоторых объектов требуется писать код, уничтожающий их явным образом (подключение в БД).Платформа .NET определяет специальный интерфейс для этих типов: **IDisposable** + оператор **using**.

**№11. Стандартные типы платформы .NET для представления коллекций данных. Итераторы.**

1. *списков*: + пользовательское коллекции Collection<T>
   1. **List<T>**. Для хранения данных набора используется внутренний массив. Класс List<T> реализует интерфейсы IList<T> и IList.
   2. **LinkedList<T>** служит для представления двусвязного списка. Такой список позволяет осуществлять вставку и удаление элемента без сдвига остальных элементов. LinkedList<T> реализует интерфейсы ICollection и ICollection<T>. Каждый элемент двусвязного списка представлен объектом LinkedListNode<T>.
   3. Классы **Queue<T>** и **Stack<T>** реализуют структуры данных «очередь» и «стек» на основе массива.
2. *множеств*:
   1. **HashSet<T>** описывает множество, в котором вхождение элемента проверяется на основе хэш-кода. Кроме реализации интерфейса ISet<T>, класс HashSet<T> содержит экземплярный метод RemoveWhere() для удаления элементов, удовлетворяющих заданному предикату.
   2. **SortedSet<T>** - это множество, поддерживающее набор элементов в отсортированном порядке
3. *словарей*:
   1. **Dictionary<TKey, TValue>** – это словарь с возможностью указать тип для ключа и тип для значения
   2. **OrderedDictionary** – это слаботипизированный класс, запоминающий порядок добавления элементов
   3. **ListDictionary** использует для хранения элементов словаря не хэш-таблицу, а односвязный список. Это делает данный класс неэффективным при работе с большими наборами данных
   4. HybridDictionary – это форма словаря, использующая список для хранения при малом количестве элементов, и переключающаяся на применение хэш-функции, когда количество элементов превышает определённый порог
   5. Платформа .NET предоставляет три класса-словаря, организованных так, что их элементы всегда отсортированы по ключу:
      1. SortedDictionary<TKey,TValue>;
      2. SortedList;
      3. SortedList<TKey,TValue> (универсальная версия SortedList).

Данные классы используют разные внутренние структуры для хранения элементов словаря. Класс SortedDictionary<TKey,TValue> быстрее классов SortedList и SortedList<TKey,TValue> при выполнении вставки элементов. Но классы SortedList и SortedList<TKey,TValue> могут предоставить доступ к элементу не только по ключу, а и с использованием целочисленного индекса.

Опишем набор интерфейсов, реализуемых практически всеми типами коллекций.

* **IEnumerable<T>/IEnumerable** отражают возможность коллекции **перечислить** её **элементы**.
* **ICollection** – это интерфейс для коллекций, **запоминающих число хранимых элементов**. определяет свойство Count, а также метод для копирования коллекции в массив и свойства для синхронизации.
* Универсальный интерфейс **ICollection<T>** также поддерживает свойство для количества элементов. Кроме этого, он предоставляет методы **для добавления и удаления элементов**, **копирования элементов в массив**, **поиска элемента**.
* Интерфейс **IList/ List<T>** описывает набор данных, которые проецируются на массив. Дополнительно к позволяет обращаться к элементу **по индексу, добавлять, удалять и искать элементы**.
* Для работы с коллекциями-множествами - ISet<T> (отражает типичные операции для множеств).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Итератор** – это метод, свойство или индексатор, содержащий один или несколько операторов **yield**. Итератор должен возвращать один из следующих интерфейсов: IEnumerator, IEnumerator<T>, IEnumerable, IEnumerable<T>. Возможности итераторов широко используются в технологии LINQ. Итераторы реализуют **концепцию отложенных вычислений**. Каждое выполнение оператора yield return ведёт к выходу из метода и возврату значения. Но состояние метода, его внутренние переменные и позиция yield return запоминаются, чтобы быть восстановленными при следующем вызове. При каждом появлении оператора **yield** управление передается вызывающему коду, но состояние вызванного метода запоминается, чтобы он мог продолжить выполнение, когда будет запрошен очередной элемент. Оператор **yield return <выражение>** возвращает следующее значение последовательности, а оператор **yield break** прекращает генерацию последовательности.

**№12. ТЕХНОЛОГИЯ LINQ. ЛЯМБДА-ВЫРАЖЕНИЯ. АНОНИМНЫЕ КЛАССЫ.**

По типу обрабатываемой информации LINQ делится на: LINQ to Objects – библиотеки для обработки коллекций объектов в памяти; LINQ to SQL – библиотеки для работы с базами данных; LINQ to XML предназначена для обработки XML-информации. **LINQ to Objects** – это набор классов, содержащих типичные методы обработки коллекций: поиск данных, сортировка, фильтрация.

1. **Оператор условия Where**. Производит фильтрацию коллекции, основываясь на параметре-предикате. IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, bool> predicate);

IEnumerable<T> Where<T>(this IEnumerable<T> source, Func<T, int, bool> predicate);

*Пример*: List<int> lst = new List<int> { 1, 3, -1, -4, 7 }; var r1 = lst.Where(x => x < 0);

1. **Операторы проекций**. Выборка информации, при этом они могут изменять тип элементов итоговой коллекции. Основным оператором проекции является **Select()**: IEnumerable<S> Select<T, S>(this IEnumerable<T> source, Func<T, S> selector); Оператор **SelectMany()** может применяться в том случае, если результатом проекции является набор данных. В этом случае оператор соединяет все элементы набора в одну коллекцию. IEnumerable<S> SelectMany<T, S>(this IEnumerable<T>source, Func<T, IEnumerable<S>> selector); *Примеры*: IEnumerable<string> r1 = gr.Select(s => s.Name); var r2 = gr.Select(s => new {s.Name, s.Age}); IEnumerable<int> r3 = gr.SelectMany(s => s.Marks);
2. **Операторы упорядочивания** выполняют сортировку коллекций. Операторы **OrderBy**() и **OrderByDescending**() выполняют сортировку по возрастанию или убыванию. IOrderedEnumerable<T> OrderBy<T, K>(this IEnumerable<T> source, Func<T, K> keySelector); IOrderedEnumerable<T> OrderByDescending<T, K>(this IEnumerable<T> src, Func<T, K> keySelector); Если после выполнения сортировки по одному ключу требуется дополнительная сортировка по другому ключу, нужно воспользоваться операторами **ThenBy**() и **ThenByDescending**(). Имеется также оператор **Reverse**(), обращающий коллекцию. Пример: var r1 = Enumerable.Reverse(gr); var r2 = gr.OrderBy(s => s.Age);

var r3 = gr.OrderByDescending(s => s.Age).ThenBy(s => s.Name);

1. **Оператор группировки GroupBy** разбивает коллекцию на группы элементов с одинаковым значением некоторого ключа. IEnumerable<IGrouping<K, T>> GroupBy<T, K>(this IEnumerable<T> src, Func<T, K> keySelector);
2. **Операторы соединения** применяются, когда требуется соединить две коллекции, элементы которых имеют общие атрибуты
3. **Операторы работы с множествами.** Оператор Distinct() удаляет из коллекции повторяющиеся элементы. Операторы Union(), Intersect() и Except() представляют объединение, пересечение и разность двух множеств.
4. **Операторы агрегирования**. результатом работы является скалярное значение: int **Count<T>(**this IEnumerable<T> source); long **LongCount<T>(**this IEnumerable<T> source); Num **Sum<T>(**this IEnumerable<T> source, Func<T, Num> selector); Num **Average<T>(**this IEnumerable<T> source, Func<T, Num> selector); Num **Min<T>/Max<T>**(this IEnumerable<T> src, Func<T, Num> selector); T **Min<T>/Max<T>(**this IEnumerable<T> source); S **Min<T, S>/Max<T, S>(**this IEnumerable<T> src, Func<T, S> selector); T **Aggregate<T>(**this IEnumerable<T> source, Func<T, T, T> func);
5. **Операторы генерирования**. Позволяет создать набор данных. **Range**() выдаёт указанное количество подряд идущих целых чисел, начиная с заданного значения: IEnumerable<int> Range(int start, int count);

**Repeat**() создает коллекцию, в которой указанный элемент повторяется требуемое число раз: IEnumerable<T> Repeat<T>(T element, int count); **Empty**() порождает пустое перечисление определенного типа: IEnumerable<T> Empty<T>();

1. **Операторы кванторов и сравнения**. **Any**() проверяет наличие хотя бы одного элемента в коллекции :bool Any<T>(this IEnumerable<T> source, Func<T, bool> predicate); Вторая версия оператора **Any**() просто проверяет коллекцию на непустоту bool Any<T>(this IEnumerable<T> source); **All**() возвращает true, если все элементы коллекции удовлетворяют предикату: bool All<T>(this IEnumerable<T> source, Func<T, bool> predicate); оператор Contains() проверяет, входит ли заданное значение в коллекцию: bool Contains<T>(this IEnumerable<T> source, T value); Оператор SequenceEqual() сравнивает две коллекции и возвращает true, если обе коллекции имеют одинаковую длину и их соответствующие элементы равны
2. **Операторы разбиения** выделяют некую часть исходной коллекции (например, первые десять элементов). IEnumerable<T> Take<T>(this IEnumerable<T> source, int count); IEnumerable<T> TakeWhile<T>(this IEnumerable<T> source, Func<T, bool> predicate); IEnumerable<T> Skip<T>(this IEnumerable<T> source, int count); IEnumerable<T> SkipWhile<T>(this IEnumerable<T> source, Func<T, bool> predicate);
3. **Операторы элемента в**ыделяют из коллекции элемент, удовлетворяющего определенным условиям.

First() выделяет первый элемент, удовлетворяющий определенному условию. Если в коллекции нет элементов, удовлетворяющих предикату, оператор First() выбрасывает исключение InvalidOperationException. Если требуется, чтобы исключение не выбрасывалось, а возвращалось предопределенное значение для типа, следует использовать оператор FirstOrDefault(). Аналогично работают операторы Last() и LastOrDefault() для выделения последнего элемента. Операторы Single() и SingleOrDefault() рассчитаны на то, что коллекция будет состоять из одного элемента, который данные операторы и возвращают. Если в коллекции нет элементов, или их оказалось больше одного, оператор Single() выбрасывает исключение InvalidOperationException. Оператор SingleOrDefault() выбрасывает такое исключение, если элементов оказалось больше одного. Пара операторов ElementAt() и ElementAtOrDefault() пытаются вернуть элемент на указанной целочисленной позиции. Оператор DefaultIfEmpty() проверяет коллекцию на пустоту. Если в коллекции нет элементов, то возвращается или значение по умолчанию для типа, или указанное значение. Если коллекция непустая, то она и возвращается.

1. **Операторы преобразования** преобразуют универсальные коллекции, реализующие IEnumerable<T>, в конкретные типы. Операторы **ToArray**() и **ToList**(), **ToDictionary**(). Оператор **OfType**<T>() итерируется по коллекции и генерирует список, содержащий только элементы заданного типа T. Оператор **Cast<T>()** итерируется по слаботипизированной коллекции и пытается выполнить приведение каждого элемента к указанному типу. Если приведение выполнить не удаётся, генерируется исключение.

Язык C# вводит синтаксические расширения для записи некоторых операторов LINQ в виде выражений запросов. Составленное выражение запросов должно подчиняться следующим правилам:

1. Выражение должно начинаться с конструкции from, которая указывает на обрабатываемую коллекцию.
2. Затем выражение может содержать ноль или более конструкции from, let или where. Конструкция let представляет переменную и присваивает ей значение. Конструкция where фильтрует элементы коллекции.
3. Затем выражение может включать ноль или более конструкций orderby, с полями сортировки и необязательным указанием на направление упорядочивания. Направление: ascending или descending.
4. Затем следует конструкция select или group.
5. Наконец, в оставшейся части выражения может следовать необязательная конструкция продолжения. Такой конструкцией является into. var r1 = gr.Where(s => s.Age > 20).Select(s => s.Marks); === var r2 = from s in gr where s.Age > 20 select s.Marks;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Лямбда-выражения представляют собой метод без имени, написанный вместо экземпляра делегата. Они описываются в следующей форме: (параметры)=>(выражение или операторный блок). При использовании лямбда-операторов список параметров отделяется от тела оператора символами =>, а КСлово delegate не указывается.

delegate int Transformer (int i); delegate int Transformer (int i);

class Test { class Test {

static void Main() { static void Main() {

Transformer square = x => x\*x; Transformer square = Square;

Console.WriteLine(square(3)); Console.WriteLine(square(3));

} }

static int Square(int x) {return x\*x;}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Анонимный тип является простым классом, созданным динамически для хранения набора значений. Для его создания служит ключевое слово new, за которым следуют {}, где указываются какие свойства и значения будет содержать тип: var dude = new {Name=”Bob”, Age = 20}; Анонимные типы применятются, в основном, принаписании LINQ-запросов.

**№13. РАБОТА С ФАЙЛАМИ И ПОТОКАМИ ДАННЫХНА ПЛАТФОРМЕ .NET**

Классы File и FileInfo предназначены для работы файлами. Первый выполняет операции при помощи статических методов, второй – при помощи экземплярных методов.

Класс FileInfo описывает файл на диске и позволяет производить операции с этим файлом. Как правило, код, работающий с данными файла, вначале вызывает метод Open(). Метод Open() имеет перегруженную версию, которая принимает три параметра:

1. определяет режим запроса на открытие файла.
2. определяет тип доступа к данным файла.
3. задаёт возможность совместной работы с открытым файлом и представлен значениями перечисления:

**Пример**: var file = new FileInfo(@"C:\Test.txt"); FileStream fs = file.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite, FileShare.None);

Кроме методов класса FileInfo, статический класс File обладает методами, позволяющими легко прочитать и записать информацию, содержащуюся в файле определенного типа.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*Поток данных* – это абстрактное представление данных в виде последовательности байт. Поток либо ассоциируется с неким физическим хранилищем (файлами на диске, памятью, сетью), либо декорирует (обрамляет) другой поток, преобразуя данные тем или иным образом. *Адаптеры потоков* служат оболочкой потока, преобразуя информацию определённого формата в набор байт.

Классы для работы с потоками:

1. System.IO.**Stream** - базовый класс для других классов, представляющих потоки. Вводит поддержку асинхронного ввода/вывода.
2. Классы для работы с потоками, связанными с хранилищами.
   1. **FileStream** – класс для работы с файлами, как с потоками (пространство имён System.IO). Объект класса FileStream возвращается некоторыми методами классов FileInfo и File. Кроме этого, данный объект можно создать при помощи конструктора с параметрами, включающими имя файла и опции доступа к файлу.
   2. **MemoryStream** – класс для представления потока в памяти (пространство имён System.IO). Метод ToArray() записывает все содержимое потока в массив байт. Метод WriteTo() переносит содержимое потока из памяти в другой поток, производный от класса Stream.
   3. **NetworkStream** – работа с сокетами, как с потокам (пространство имён System.Net.Sockets).
3. Декораторы потоков.
   1. System.IO.Compression.**DeflateStream** и **GZipStream** – классы для потоков со сжатием данных
   2. **CryptoStream** – поток зашифрованных данных (пространство имён System.Security.Cryptography).
   3. **BufferedStream** – поток с поддержкой буферизации данных (пространство имён System.IO).
4. Адаптеры потоков.
   1. **BinaryReader** и **BinaryWriter** позволяют при помощи своих методов читать и записывать в поток данные примитивных типов и массивов байт или символов. Вся информация записывается в поток в двоичном представлении.
   2. **StreamReader** и **StreamWriter** – классы для ввода/вывода информации в строковом представлении. Абстрактные классы TextReader и TextWriter дают возможность читать и записывать данные в поток в строковом представлении. От этих классов наследуются классы StreamReader и StreamWriter.
   3. **XmlReader** и **XmlWriter** – абстрактные классы для ввода/вывода XML.

**№14. ИСПОЛЬЗОВАНИЕ XML НА ПЛАТФОРМЕ .NET**

XML – это способ описания структурированных данных. Структурированными данными называются такие данные, которые обладают заданным набором семантических атрибутов и допускают иерархическое описание.

Технология LINQ to XML предоставляет программный интерфейс для работы с XML-документами, описываемыми в виде дерева объектов. Этот программный интерфейс обеспечивает создание, редактирование и трансформацию XML, при этом возможно применение LINQ-подобного синтаксиса.

LINQ_XML_2.emf

* + Классы XElement и XDocument представляют XML-элемент и XML-документ соответственно.
  + Класс XAttribute служит для описания XML-атрибута.
  + Класс XText представляет текст в XML-элементе.
  + Класс XName представляет имя атрибута или элемента.
  + Классы XDeclaration, XDocumentType, XProcessingInstruction, XComment описывают соответствующие части XML-документа.
  + Статический класс Extensions содержит методы расширения для выполнения запросов к XML-данным.

1. **Создание, сохранение, загрузка XML.** Для создания отдельного XML-элемента обычно используется один из конструкторов класса XElement: var e1 = new XElement("name", "Earth"); // <name>Earth</name>

var e2 = new XElement("planet", e1);

// <planet>

// <name>Earth</name>

// </planet>

* 1. Конструкторы класса **XDocument** позволяют указать декларацию XML-документа и набор объектов содержимого. В этот набор могут входить комментарии, инструкции по обработке и не более одного XML-элемента:
  2. Кроме применения конструкторов, объекты XML можно создать из строкового представления при помощи статических методов XElement.Parse() и XDocument.Parse(): var planet = XElement.Parse("<name>Earth</name>");
  3. Для сохранения элемента или XML-документа используется метод Save(), имеющийся у XElement и XDocument. Данный метод перегружен и позволяет выполнить запись в текстовый файл или с применением адаптеров TextWriter и XmlWriter.
  4. Загрузка элемента или XML-документа выполняется статическими методами XElement.Load() или XDocument.Load(). Метод Load() перегружен и позволяет выполнить загрузку из файла, произвольного URI, а также с применением адаптеров TextReader и XmlReader.

1. Запросы, модификация и трансформация XML
   1. Класс XObject имеет свойство NodeType для типа XML-узла и свойство Parent, указывающее, какому элементу принадлежит узел.
   2. Методы классов XNode и XContainer позволяют получить у элемента наборы предков и дочерних узлов (элементов). При этом возможно указание фильтра – имени элемента. Большинство методов возвращают коллекции, реализующие IEnumerable .
   3. Класс XDocument позволяет получить корневой элемент при помощи свойства Root. В классе XElement есть методы для запроса элементов-предков и элементов-потомков, а также методы для запроса атрибутов.
   4. Статический класс Extensions определяет несколько методов расширения, работающих с коллекциями элементов или узлов XML:
   5. **Модификация** XML-информации в памяти выполняется методов классов XNode, XContainer, XElement:
2. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ ОБРАБОТКИ XML

* В дополнение к LINQ to XML, платформа .NET содержит несколько программных интерфейсов для работы с XML. Для этого обычно используются классы из пространства имён System.Xml (и пространств вида System.Xml.\*).
* Классы XmlReader и XmlWriter - это основа механизма последовательного чтения, обработки и записи XML-документов. Такой подход выгодно использовать, когда документ слишком велик, чтобы читать его в память целиком, или содержит ошибки в структуре.
* Для последовательного чтения XML-документов применяется класс XmlReader и его наследники: XmlTextReader (чтение на основе текстового потока), XmlNodeReader (разбор XML из объектов XmlNode) и XmlValidatingReader (при чтении производится проверка схемы документа). Обычно для простого чтения XML достаточно класса XmlTextReader. Объект XmlTextReader извлекает XML-конструкции из потока при помощи метода Read(). Тип текущей конструкции можно узнать, используя свойство NodeType, значениями которого являются элементы перечисления XmlNodeType. С конструкцией можно работать, используя различные свойства, такие как Name (возвращает имя элемента или атрибута), Value (возвращает данные элемента) и так далее.
* Набор методов класса XmlTextReader, начинающихся с префикса Move (MoveToNextElement()), может использоваться для перехода к следующей XML-конструкции в потоке. Вернуться к просмотренным конструкциям нельзя.
* Класс XmlWriter – это абстрактный класс для создания XML-данных. Подчеркнем, что XML-данные всегда могут быть сформированы в виде простой строки и затем записаны в любой поток. Однако такой подход не лишен недостатков возрастает вероятность неправильного формирования структуры XML из-за элементарных ошибок программиста.
* Классы XmlNode, XmlAttribute, XmlElement, XmlDocument служат для представления XML-документа в виде дерева объектов. Программный интерфейс, основанный на использовании данных классов, являлся предшественником LINQ to XML.

**№15. СОСТАВ И ВЗАИМОДЕЙСТВИЕ СБОРОК**

Сборка – это единица развёртывания и контроля версий. Сборка состоит из следующих компонетов:

* *манифест сборки* – предоставляет платформе такую информацию как имя сборки, её версию, список других сборок, на которые она ссылается.
* *манифест приложения* – предоставляет ОС информацию о том, как должно происходит развертывание сборки и требуется ли повышение прав;
* *откомпилированные типы* – IL-код и метаданные типов, определённые в сборке.
* *ресурсы* – неисполняемые данные (изображения, текст для локализации и т.п.)

Из этих компонентов только манифест является обязательным. Будем называть сборку однофайловой, если она состоит из одного файла. В противном случае сборку будем называть многофайловой.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Рассмотрим вопрос взаимодействия сборок. Как правило, крупные про-граммные проекты состоят из нескольких сборок, связанных ссылками. Среди этих сборок имеется некая основная (обычно оформленная как исполняемый файл \*.exe), а другие сборки играют роль подключаемых библиотек с кодом необходимых типов (обычно такие сборки – это файлы с расширением \*.dll).

Платформа .NET разделяет сборки на:

* *локальные*. При выполнении приложения она должна находиться в том же каталоге, что и main.exe. Локальные сборки обеспечивают простоту развёртывания приложения (все его компоненты сосредоточены в одном месте) и изолированность компонентов. Имя локальной сборки – слабое имя – это имя файла сборки без расширения
* *глобальные* (или сборки с сильными именами). Сборки, помещаемые в специальное защищенное хранилище сборок GAC, должны удовлетворять определённым условиям:
  + - * такие глобальные сборки должны иметь цифровую подпись. Это исключает подмену сборок злоумышленниками.
      * для глобальных сборок отслеживаются версии и языковые культуры.
      * Сборка, помещенная в GAC, получает сильное имя, которое включает: имя главного файла сборки (без расширения), версию сборки, указание о региональной принадлежности сборки и маркер открытого ключа сборки:

Рассмотрим процесс создания сборки с сильным именем на примере сборки UL.dll.

1. необходимо создать пару криптографических ключей для цифровой подписи сборки. Для этих целей служит утилита sn.exe, входящая в состав Microsoft .NET Framework SDK
2. Далее необходимо подписать сборку полученными ключами. Для этого используется специальный атрибут уровня сборки [AssemblyKeyFile] или ключ компилятора командной строки /keyfile:
3. сборку можно поместить в GAC. Простейший вариант сделать это – использовать утилиту gacutil.exe, входящую в состав .NET Framework SDK. При использовании ключа /i сборка помещается в GAC, а ключ /u удаляет сборку из GAC:

Если программист желает указать версию, то для этого используется атрибут [AssemblyVersion]. Номер версии имеет формат Major.Minor.Build.Revision. Часть Major является обязательной. Любая другая часть может быть опущена (в этом случае она полагается равной нулю).

**№16. МЕТАДАННЫЕ И ИНФОРМАЦИЯ О ТИПАХ. ТЕХНОЛОГИЯ «ОТРАЖЕНИЯ».**

При создании сборки в неё помещаются *метаданные*, которые являются описанием всех типов в сборке и их элементов. Программист может работать с метаданными, используя специальный механизм, называемый *отражением*.

Использование возможностей отражения:

* класс System.**Type**. Служит для получения информации о типе: имя типа, имя базового типа, является ли тип универсальным, в какой сборке он размещается и другую информацию. Кроме этого, Type имеет специальные методы, возвращающие данные о полях типа, свойствах, событиях, методах и их параметрах. Способы получения объекта этого класса:
  + 1. Вызвать у объекта GetType(), который определен на уровне System.Object.
    2. Использовать статический метод Type.GetType(), которому передается имя типа в виде строки (имя должно быть полным, то есть включать пространство имён).
    3. Использовать операцию C# typeof, параметром которой является тип.
* типы из пространств имён System.Reflection и System.Reflection.Emit. Пространство имен System.**Reflection** содержит типы для получения информации и манипулирования сборкой и модулем сборки. При помощи класса Assembly можно получить информацию о сборке, при помощи класса Module о модуле.

Механизм отражения позволяет реализовать на платформе .NET **позднее связывание** (late binding). Этот термин обозначает процесс динамической загрузки типов при работе приложения, создание экземпляров типов и работу с элементами экземпляров. Для этого:

1. Надо загрузить в память сборки с типом – выполняется при помощи метода Assembly.Load(). Для указания имени сборки можно использовать простую строку или объект класса AssemblyName.
2. Надо создать объект требуемого типа. Для этого следует воспользоваться методом Assembly.CreateInstance() .

**№17. МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ. СИНХРОНИЗАЦИЯ ПОТОКОВ ВЫПОЛНЕНИЯ (НА ПРИМЕРЕ ПЛАТФОРМЫ .NET)**

Основные классы, предназначенные для поддержки многопоточности, сосредоточены в пространстве имен System.**Threading**. На платформе .NET каждый поток выполнения (thread) представлен объектом класса **Thread**.

* Для организации собственного потока необходимо создать объект класса **Thread**.
* Создание потока не подразумевает его автоматического запуска. Для запуска потока требуется вызвать метод Start().

*Основные свойства* класса Thread:

* CurrentThread возвращает объект, представляющий текущий поток.
* ManagedThreadId возвращает уникальный числовой идентификатор управляемого потока.
* ThreadState позволяет получить текущее состояние потока.
* IsAlive позволяет определить, выполняется ли поток.
* Priority управляет приоритетом выполнения потока относительно текущего процесса. Значением этого свойства являются элементы перечисления ThreadPriority: Lowest, BelowNormal, Normal, AboveNormal, Highest.
* IsBackground позволяет сделать поток фоновым.

*Основные методы* класса Thread:

* Suspend() вызывает приостановку потока
* Resume() возобновляет работу потока
* Sleep() приостанавливает выполнение текущего потока (можно указать количество миллисекунд или значение TimeSpan)
* Для завершения работы выбранного потока используется метод Abort().

.NET поддерживает специальный механизм, называемый **пул потоков**, который состоит из двух основных элементов: очереди методов и рабочих потоков. Характеристикой пула является его ёмкость – **максимальное число рабочих потоков**. При работе с пулом метод сначала помещается в очередь. Если у пула есть свободные рабочие потоки, метод извлекается из очереди и направляется свободному потоку для выполнения. Если свободных потоков нет, но ёмкость пула не достигнута, для обслуживания метода формируется новый рабочий поток. Однако этот поток создается с задержкой в полсекунды. Если за это время освободится какой-либо из рабочих потоков, то он будет назначен на выполнение метода, а новый рабочий поток создан не будет. Важным нюансом является то, что несколько первых рабочих потоков в пуле создаётся без полусекундной задержки. Для работы с пулом используется статический класс **ThreadPool**. Метод **SetMaxThreads**() позволяет изменить ёмкость пула, которая по умолчанию равна 1023. Метод **SetMinThreads**() устанавливает количество рабочих потоков, создаваемых без задержки. По умолчанию их число равно количеству процессорных ядер. Для помещения метода в очередь пула служит метод **QueueUserWorkItem**().

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.

*Синхронизация потоков* – это координирование действий потоков для получения предсказуемого результата. Средства синхронизации потоков можно разделить на четыре категории:

1. простые методы приостановки выполнения потока (Suspend(), Resume(), Sleep(), Yield(), Join());
2. блокирующие конструкции обеспечивают исключительный доступ к ресурсу. Блокировка позволяет потокам работать с общими данными, не мешая друг другу. Для организации блокировок платформа .NET предоставляет такие классы, как Monitor, Mutex, Semaphor, SemaphorSlim, а язык C# - оператор lock.
3. конструкции подачи сигналов;
4. не задерживающие средства синхронизации: класс System.Threading.**Interlocked**, который имеет методы для инкремента, декремента и сложения аргументов типа int или long, а также методы присваивания значений числовым и ссылочным переменным. Каждый метод выполняется как атомарная операция.

Первые три категории используют для синхронизации приостановку потока. Приостановленный поток практически не потребляет времени процессора, однако его выполнение легко возобновляется системой при наступлении условия «пробуждения».

**№18. ШАБЛОНЫ ПРОЕКТИРОВАНИЯ. АНТИПАТТЕРНЫ.**

Шаблоны проектирования – это многократно используемые решения распространенных проблем, возникающих при разработке программного обеспечения. Зная эти шаблоны, опытные программисты распознают ситуацию их применения и сразу используют готовое решение, не тратя времени на предварительный анализ проблемы.

Главная польза каждого отдельного шаблона состоит в том, что он описывает решение целого класса абстрактных проблем. Тот факт, что каждый шаблон имеет свое имя, облегчает дискуссию между разработчиками. Таким образом, за счёт шаблонов **производится унификация терминологии**, названий модулей и элементов проекта.

Однако есть мнение, что слепое применение шаблонов из справочника, без осмысления причин и предпосылок выделения шаблона, **замедляет профессиональный рост программиста**, так как подменяет творческую работу механической подстановкой. Люди, придерживающиеся данного мнения, считают, что знакомиться со списками шаблонов следует тогда, когда программист «дорос» до них в профессиональном плане.

Классификация шаблонов **в зависимости от назначения**:

* *Структурные* – показывают, как объекты и классы объединяются для образования сложных структур. Примеры: Adapter, Bridge, Decorator.
* *Порождающие* – управляют и контролируют процесс создания и жизненный цикл объектов. Примеры: Abstract Factory, Singleton, Factory Method.
* *Поведения* – используются для организации, управления и объединения различных вариантов поведения объектов. Примеры: State, Observer, Iterator.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Антипаттерны** - это классы наиболее часто внедряемых плохих решений проблем. Они изучаются в случае, когда их хотят избежать в будущем, и некоторые отдельные случаи их могут быть распознаны при изучении неработающих систем. Это как бы противоположность шаблонам, представляющим хорошие методы программирования.

* **Ненужная сложность** - внесение ненужной сложности в решение. Ненужная сложность не обусловлена действительной сложностью решаемой проблемы, а искусственно внесена при разработке архитектуры и дизайна ПО.
* **Действие на расстоянии** - неожиданное взаимодействие между широко разделёнными частями системы. Антипаттерн может проявиться при использовании в программе глобальных переменных.
* **Слепая вера** - недостаточная проверка корректности исправления ошибки или результата работы подпрограммы.
* **Лодочный якорь** - сохранение более не используемой части системы.
* **Таинственный код** - использование аббревиатур вместо мнемоничных имён.
* **Магические числа** - включение чисел в алгоритмы без объяснений.
* **Синглетонизм** - избыточное использование шаблона Одиночка.

**№19. СТРУКТУРНЫЕ ШАБЛОНЫ ПРОЕТИРОВАНИЯ.**

Показывают, как объекты и классы объединяются для образования сложных структур.

**Декоратор** (Decorator). Шаблон даёт способ для динамического добавления к объекту нового состояния и поведения. При этом исходный объект не знает о том, что он «декорируется». Ключевым моментом реализации шаблона является то, что класс-декоратор одновременно наследуется от декорируемого класса и агрегирует объект этого класса. На диаграмме и декоратор, и декорируемый объект реализуют общий интерфейс. Возможен вариант, когда вместо интерфейса используется общий класс-предок. Кроме этого, декоратор агрегирует декорируемый объект, добавляя к нему новое поведение.

decorator.emf

Обратите внимание на то, что несколько декораторов независимо могут применяться к одному и тому же объекту, а также на то, что декоратор может декорировать объект, который уже декорирован.

**Мост** (Bridge). Шаблон Мост используется в тех случаях, когда имеется отдельная иерархия абстрактных классов и интерфейсов и соответствующая иерархия реализаций. Мост соединяет абстракции и реализации в виде независимых динамических классов. Выбор конкретной реализации может выполняться по-разному, например, на основе данных в файле конфигурации. На диаграмме показан только один абстрактный тип. Этот тип агрегирует некий конкретный объект-мост, которому делегируется выполнение требуемых операций.

Bridge.emf

**Адаптер** (Adapter). Шаблон Адаптер предназначен для обеспечения совместной работы классов, которые изначально не были предназначены для совместного использования. Такие ситуации часто возникают, когда идет работа с существующими библиотеками кода. Нередко их интерфейс не отвечает требованиям клиента, но изменить библиотеку возможности нет. Возникает задача адаптации библиотеки для клиента. В адаптере четко видно преимущество программирования согласно интерфейсам.

Adapter.emf

Особенность адаптеров в том, что они могут добавлять дополнительное поведение к тому поведению, что специфицируется в ITarget и в Adaptee. Другими словами, адаптеры могут быть прозрачными для клиента и непрозрачными

**Фасад** (Façade). Назначение шаблона Фасад состоит в предоставлении различных высокоуровневых представлений подсистем, детали реализации которых скрыты от клиента. В общем случае набор операций, желаемый для клиента, может формироваться в виде набора из разных частей подсистемы.

В качестве иллюстрации использования шаблона Фасад приведём работу с классами подсистемы отправки почтовых сообщений. Письмо, адресат, тело письма, почтовый сервер, присоединённые к письму файлы – все это является отдельным объектами специальных классов. Без применения фасада клиент должен взаимодействовать со всем перечисленным набором объектов, используя их свойства и методы. Фасад берет работу по взаимодействию на себя теперь клиент использует только объект-фасад, который делегирует работу нужным подсистемам.

Сокрытие деталей – это ключевая концепция программирования. Шаблон Фасад отличает от таких шаблонов, как Декоратор или Адаптер, тот факт, что интерфейс, который строится над подсистемой, может быть абсолютно любым, он не должен удовлетворять заданным заранее правилам. Допускается существование даже нескольких фасадов для одного набора подсистем.

Детали реализации шаблона Фасад не представляют трудностей. Из возможных вариантов отметим прозрачные фасады (в этом случае компоненты подсистемы могут быть доступны и через фасад, и в обход его) и статические фасады (фасад является статическим классом – скрываемые объекты не агрегируются, а создаются в методах фасада по необходимости).

facade.emf

**№20. ПОРОЖДАЮЩИЕ ШАБЛОНЫ ПРОЕКТИРОВАНИЯ.**

Управляют и контролируют процесс создания и жизненный цикл объектов.

**Прототип** (Prototype). Шаблон позволяет создавать специальные объекты, ничего не знаю об их классе или деталях создания. Механизм можно описать так: клиенту, инициирующему создание объектов, предоставляются объекты-прототипы. Затем целевые объекты создаются путем клонирования этих объектов-прототипов.

Объект обычно создаётся путем вызова конструктора. Если же используется шаблон Прототип, то клиент знает только об интерфейсе IPrototype, реализующем операции клонирования объекта. Реальный класс объекта клиенту не известен. Прототипы для клонирования могут использоваться произвольное количество раз, сами он при операции клонирования менятся не должны. Хотя существуют различные варианты дизайна данного шаблона, наиболее гибким является вариант с менеджером прототипов, содержащим индексированный список доступных прототипов.

Prototype.emf

**Фабричный метод** (Factory method). Занимается созданием объектов. Каждый такой объект принадлежит некому классу, однако все классы либо имеют общего предка, либо реализуют общий интерфейс. Фабричный метод сам решает, какой конкретный класс нужно использовать для создания очередного объекта. Решение принимается либо на основе информации, предоставленной клиентом, либо на основе внутреннего состояния метода.

factoryMethod.emf

**Одиночка** (Singleton). Гарантирует создание единственного экземпляра объекта некоторого класса.Шаблон Одиночка добавляет функциональность путём модификации существующего класса. Модификация требует следующих шагов.

1. Конструктор класса делается закрытым (private).
2. Добавляется закрытое статическое поле только для чтения.
3. Для доступа к закрытому статическому полю добавляется открытое статическое свойство.

Singleton.emf

**№21. ПАТТЕРНЫ ПОВЕДЕНИЯ.**

**Стратегия** (Strategy). *Назначение*: определяет семейство алгоритмов, инкапсулирует каждый их них и обеспечивает их взаимозаменяемость. Он позволяет модифицировать алгоритм независимо от их использования на стороне клиента. *Применимость*:

* Имеется много родственных классов, отличающихся лишь поведением.
* Надо иметь несколько вариантов алгоритма.
* В алгоритме есть данные, о которых клиент не должен знать.
* В классе определено много поведений, что представлено разветвленными условными операторами.

Strategy.emf

**Состояние** (State). Шаблон Состояние можно рассматривать как динамическую версию шаблона Стратегия. При использовании шаблона Состояние поведение объекта меняется в зависимости от текущего контекста. При помощи шаблона Состояние можно эффективно реализовать такую абстракцию как конечный автомат.

При наивной программной реализации конечного автомата порождаются наборы конструкций switch-case, которые, как правило, вложены в друг друга. Использование шаблона Состояние позволяет упростить код. Все состояния конечного автомата описываются отдельными классами, которые обладают набором виртуальных методов, соответствующих входным сигналам. Получение очередного входного сигнала означает вызов метода того объекта, экземпляр которого находится в поле state. При этом сам вызываемый метод может поместить в state другой объект-состояние (переключить состояние).

State.emf

При практической реализации шаблона Состояние отдельные объекты-состояния могут быть оформлены с применением шаблона Одиночка и получать объект-контекст в качестве одно из параметров своих методов.

**Итератор** (Iterator). Назначение шаблона Итератор – предоставить последовательный доступ к элементам коллекции без информации о её внутреннем устройстве. Дополнительно шаблон может реализовывать функционал по фильтрации элементов коллекции. В последние годы возможности создания итераторов и перечислителей были интегрированы в синтаксис многих языков программирования, что позволило упростить код и сделать реализацию шаблонов компактной. Диаграмма шаблона Итератор использует имена, применяемые в .NET Framework и языке C#.

Iterator.emf

При практической реализации шаблона Итератор на C#, безусловно, используются такие возможности, как перечислители, интерфейсы IEnumerable и IEnumerator, оператор yield.

**Наблюдатель** (Observer). Шаблон Наблюдатель определяет отношение между объектами таким образом, что когда один из объектов меняет своё состояние, все другие объекты получают об этом уведомление. Иллюстрацией применения шаблона Наблюдатель служит знакомая по языку C# система событий. Один объект публикует событие, остальные объекты могут подписаться на событие и получать уведомление о его наступлении. Собственно, основное назначение шаблона Наблюдатель – это реализация системы работы с событиями.

Observer.emf

Как и в случае с шаблоном Посредник, дизайн шаблона Наблюдатель предполагает наличие двух выделенных классов. Объект класса Subject изменяет своё состояния, и именно эти изменения предполагается отслеживать. Объекты (их может быть несколько) класса Observer могут подписываться на отслеживание изменений. Класс Subject располагает закрытым событием с именем Notify. Как только Subject изменяет своё состояние, событие активируется. При этом вызывается метод Update() подписчиков, которому передаётся состояние объекта Subject. Метод Update() предварительно регистрируется в Subject при помощи операции Attach().

**№22. КОНЕЧНЫЕ АВТОМАТЫ – ОПИСАНИЕ, НАЗНАЧЕНИЕ, ВОЗМОЖНОСТИ. РАЗЛИЧНЫЕ СПОСОБЫ РЕАЛИЗАЦИИ**

**Конечным автоматом** (автоматом Мили) называется шестёрка объектов A=〈S,X,Y,s\_0,δ,λ〉, где: S конечное непустое множество (состояний); X конечное непустое множество входных сигналов (входной алфавит); Y конечное непустое множество выходных сигналов (выходной алфавит); s\_0∈S начальное состояние; δ: S×X⟶S функция переходов; λ: S×X⟶Y функция выходов.

Конечные автоматы широко применяются при решении различных задач математики и программирования (бот, программу-переговорщик (negotiator), интерпретатор протокола связи).

Способы реализации: графы, таблица переходов.

|  |  |  |
| --- | --- | --- |
| Состояние | 0 | 1 |
| 0 | 0 | 1 |
| 1 | 2 | 3 |
| 2 | 4 | 0 |
| 3 | 0 | 2 |
| 4 |  | 4 |

Шаблон Состояние. При использовании шаблона Состояние поведение объекта меняется в зависимости от текущего контекста. Все состояния конечного автомата описываются отдельными классами, которые обладают набором виртуальных методов, соответствующих входным сигналам. Получение очередного входного сигнала означает вызов метода того объекта, экземпляр которого находится в поле state. При этом сам вызываемый метод может поместить в state другой объект-состояние (переключить состояние). При практической реализации шаблона Состояние отдельные объекты-состояния могут быть оформлены с применением шаблона Одиночка и получать объект-контекст в качестве одно из параметров своих методов.

State.emf

Конечные автоматы подразделяются на детерминированные и недетерминированные.

Детерминированным конечным автоматом (ДКА) называется такой автомат, в котором для каждой последовательности входных символов существует лишь одно состояние, в которое автомат может перейти из текущего.

Недетерминированный конечный автомат (НКА) является обобщением детерминированного.

Другие способы описания конечного автомата:

Диаграмма состояний (или иногда граф переходов) — графическое представление множества состояний и функции переходов. Представляет собой нагруженный однонаправленный граф, вершины которого — состояния КА, ребра — переходы из одного состояния в другое, а нагрузка — символы, при которых осуществляется данный переход. Если переход из состояния q1 в q2 может быть осуществлен при появлении одного из нескольких символов, то над дугой диаграммы (ветвью графа) должны быть надписаны все они.

Таблица переходов — табличное представление функции δ. Обычно в такой таблице каждой строке соответствует одно состояние, а столбцу — один допустимый входной символ. В ячейке на пересечении строки и столбца записывается действие, которое должен выполнить автомат, если в ситуации, когда он находился в данном состоянии на входе он получил данный символ.

**№23. АРХИТЕКТУРА ПРИЛОЖЕНИЙ И ПРОГРАММНЫХ ПРОЕКТОВ. ПРИМЕРЫ ТИПОВЫХ АРХИТЕКТУР**

**Архитектура ПО** – это представление, которое дает информацию о компонентах ПО, обязанностях отдельных компонент и правилах организации взаимосвязей между компонентами. Продуманная архитектура облегчает разработку и дальнейшее развитие ПО.

1. **«Клиент-сервер».** Главными преимуществами стиля «клиент-сервер» являются:
   1. Высокая безопасность. Все данные хранятся на сервере, обеспечивающем больший уровень безопасность, нежели отдельный клиент.
   2. Централизованный доступ к данным. Так как данные хранятся только на сервере, ими легко управлять (например, обеспечить обновление).
   3. Легкость сопровождения. Роль сервера могут выполнять несколько физических компьютеров, объединенных в сеть. Благодаря этому клиент не замечает сбоев или замены отдельного серверного компьютера.
2. **Многоуровневая архитектура**. Сосредоточена на иерархическом распределении отдельных частей системы при помощи эффективного разделения отношений. Каждая часть соотносится с определенным уровнем (layer), для каждого уровня заданы выполняемые им функции, уровни выстроены в стековую структуру (то есть находятся один поверх другого).
3. **Шина сообщений**. Подразумевает наличие общего коммуникационного канала, используя который компоненты обмениваются информацией. Компонент помещает сообщение в коммуникационный канал, после этого сообщение рассылается всем заинтересованным компонентам. Данная архитектура направлена на решение коммуникационных задач. Принципы:
   1. Все коммуникации между компонентами выполняются только при помощи информационных сообщений.
   2. Сообщения имеют стандартный формат. Это позволяет интегрировать компоненты, разработанные на разных платформах.
   3. Общая логика изменяется путем удаление или добавления компонент, подключенных к шине.
4. **Объектно-ориентированная архитектура**. Система воспринимается как набор взаимодействующих объектов. Каждый такой объект содержит данные и необходимое поведение, а коммуникация между объектами происходит через открытые интерфейсы и путем посылки и приема сообщений.
5. **Архитектура, ориентированная на сервисы**. Предоставляет требуемые функции в виде набора сервисов. Сервисы используют стандартные протоколы для вызова своих функций, публикации в сети и обнаружения. Отдельный сервис должен рассматриваться как независимое приложение, не как компонент или объект. Основной задачей при использовании SOA является определение интерфейса сервиса и схемы передаваемых при вызове сервиса данных. Базовые принципы SOA:
   1. Сервисы автономны и независимы друг от друга.
   2. Сервисы распределены в локальной или глобальной сети. Местоположение сервиса не важно – главное, чтобы сеть доступа к сервису поддерживала требуемые протоколы.
   3. Сервисы публикуют контракты использования и схемы для данных обмена, но скрывают внутренние классы своей реализации.

**№24. ТЕХНОЛОГИЯ WPF. ОБЩЕЕ ОПИСАНИЕ.**

WPF – технология для построения пользовательского интерфейса, являющаяся частью платформы .NET. WPF разработана как альтернатива технологии Windows Forms, которая базируется на стандартном системном программном интерфейсе для работы с элементами управления.

* + 1. **Собственные методы построения и отрисовки элементов**. В Windows Forms классы для элементов управления делегируют функции отображения системным библиотекам, таким как user32.dll. В WPF любой элемент управления полностью строится (рисуется) самой WPF. Для аппаратного ускорения отрисовки применяется технология DirectX.
    2. **Независимость от разрешения**. WPF ориентирована на использование векторных примитивов, что делает эту технологию независимой от разрешения монитора. В WPF используется особая единица измерения, равная 1/96 дюйма.
    3. **Декларативный пользовательский интерфейс**. В WPF визуальное содержимое отображаемого окна можно полностью описать в виде документа XAML. XAML – это язык разметки, основанный на XML. Так как описание интерфейса отделено от кода, графические дизайнеры могут использовать профессиональные инструменты, чтобы редактировать файлы XAML, улучшая внешний вид всего приложения. Применение XAML является предпочтительным, но не обязательным – приложение WPF можно конструировать, используя только код.
    4. **Веб-подобная модель компоновки**. WPF поддерживает гибкий визуальный поток, размещающий элементы управления на основе их содержимого. В результате получается пользовательский интерфейс, который может быть адаптирован для отображения высокодинамичного содержимого.
    5. **Стили и шаблоны.** Стили стандартизируют форматирование и позволяют повторно использовать его по всему приложению. Шаблоны дают возможность изменить способ отображения любых элементов управления, даже таких основополагающих, как кнопки или поля ввода.
    6. **Анимация.** В WPF анимация неотъемлемая часть программного каркаса. Анимация определяется декларативными дескрипторами, и WPF запускает её в действие автоматически.
    7. **Приложения на основе страниц.** В WPF можно строить браузер-подобные приложения с кнопками навигации, которые позволяют перемещаться по коллекции страниц. Кроме этого, специальный тип WPF-приложения – XBAP может быть запущен внутри браузера.

**№25. ЯЗЫК XAML.**

**XAML** – это язык разметки, предлагающий основанный на XML синтаксис для представления логического дерева объектов .NET. Существует несколько подмножеств XAML:

* WPF XAML включает элементы, описывающие содержимое WPF вроде векторной графики и элементов управления;
* XPS XAML часть WPF XAML, определяющая XML-представление форматированных электронных документов. Эта часть опубликована как отдельный стандарт XML Paper Specification (XPS);
* Silverlight XAML подмножество WPF XAML, предназначенное для Silverlight-приложений. Silverlight это межплатформенный браузерный подключаемый модуль, позволяющий создавать веб-содержимое с двумерной графикой, анимацией, аудио и видео;
* WF XAML включает элементы, описывающие содержимое Windows Workflow Foundation.

Документ XAML записан в формате XML. Это означает, что имена элементов XAML чувствительны к регистру, нужна правильная вложенность элементов, некоторые символы требуют особого обозначения (например, &amp; это символ &).

Объектные элементы XAML описывают объект некоторого типа .NET (класса или структуры). Имя объектного элемента совпадает с именем типа. Необходимо, чтобы тип обладал открытым конструктором без параметров. Типы .NET обычно вложены в пространства имён. В XAML пространство имён .NET ставится в соответствие пространству имён XML. При необходимости указывается сборка, содержащая пространство имён:

xmlns:префикс="clr-namespace:ПространствоИмён;assembly=ИмяСборки"

Описание объекта подразумевает задание значений его свойств. В XAML для этого применяются атрибуты, элементы свойств и содержимое элемента. При использовании атрибутов указывается имя свойства и значение свойства в виде строки. Анализатор XAML применяет для преобразования строки в значение свойства специальные конвертеры типов. Набор стандартных конвертеров достаточно богат. При необходимости можно разработать собственный конвертер, используя базовый класс TypeConverter. Элемент свойства – это дочерний элемент объектного элемента, имеющий вид <TypeName.Property>. Содержимое элемента свойства рассматривается как значение свойства.

Механизм расширений разметки (markup extensions) позволяет вычислять значение свойства при выполнении приложения. Встретив в XAML расширение разметки, анализатор генерирует код, который создаёт объект расширения разметки и вызывает особый метод объекта для получения значения.

Анализатор XAML генерирует код, выполняющий по документу XAML создание и настройку объектов. Действия с объектами описываются в отдельном классе кода. Чтобы сослаться на объект в коде, объект должен иметь имя. Для этого в пространстве имён анализатора XAML определён атрибут Name. Чтобы связать класс кода с документом XAML используется атрибут Class из пространства имён анализатора XAML. Этот атрибут применяется только к корневому элементу и содержит имя класса, являющегося наследником класса корневого элемента:

<Window x:Class="WpfApplication1.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

<Button x:Name="okButton" Content="Click me!" />

</Window>

WPF XAML описывает логическое дерево элементов. Наряду с этим термином используется понятие визуального дерева элементов. Визуальное дерево составляют отображаемые объекты. Некоторые одиночные логические объекты распадаются на несколько визуальных составляющих, так как стоятся из нескольких визуальных примитивов. Например, любое окно Window включает визуальный примитив Border, который содержит примитив AdornerDecorator c объектами ContentPresenter и AdornerLayer.

**№26. ОСНОВНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ WPF. КОМПОНОВКА**

Формально, к элементам управления относятся классы, унаследованные от класса Control.

1. **Элементы управления содержимым.** Это элементы, допускающие размещение единственного дочернего элемента, представляющего их содержимое. Элементы управления содержимым можно разбить на три подгруппы:
   1. **Кнопки**. Все кнопки наследуются от абстрактного класса ButtonBase. Этот класс содержит событие Click, булево свойство IsPressed и свойство ClickMode оно определяет момент генерирования Click.
   2. **Простые контейнеры** предназначены для обрамления дочернего содержимого. Один из простых контейнеров элемент **Label**. **Tooltip** – всплывающая подсказка. ЭУ **Frame** предназначен для изолирования своего содержимого от остальной части пользовательского интерфейса. Этот элемент напоминает аналог из HTML.
   3. **Контейнеры с заголовком.** Элемент управления Expander напоминает GroupBox, но содержит в заголовке кнопку, которая позволяет спрятать и показать содержимое контейнера.
2. **Списковые ЭУ** могут хранить и представлять коллекцию элементов списка. Для большинства списков доступна гибкая настройка внешнего вида, а также функции фильтрации, группировки и сортировки данных, однако в данном параграфе рассматриваются только простейшие аспекты работы со списками. Подгруппы:
   1. **Селекторы**. Селекторы допускают индексацию и выбор элементов списка. ЭУ **ListBox** отображает список и допускает множественный выбор элементов. Класс **ComboBox** позволяет выбрать один элемент из списка, отображая текущий выбор и раскрывая список элементов по требованию. Отдельный элемент ComboBox представлен объектом. Элемент управления **ListView** унаследован от ListBox. Этот список позволяет настроить свой внешний вид при помощи свойства View с типом ViewBase. Элемент управления **TabControl** это простой набор страниц с закладками.
   2. **Меню**. Класс Menu может содержать коллекцию любых объектов, но ожидается, что будут использованы объекты MenuItem и Separator. Separator – простой элемент, представляющий разделитель пунктов меню. MenuItem – это контейнер с заголовком. MenuItem поддерживает «горячие клавиши».
3. **Прочие элементы управления.**
   1. **Текстовые ЭУ**. **TextBlock** предназначен для отображения небольшой порции текста. RichTextBox – «продвинутая» версия **TextBox**. **PasswordBox** предназначен для ввода паролей.
   2. **Элементы для представления диапазона** ProgressBar и Slider хранят и отображают в некой форме числовое значение, попадающее в заданный диапазон. ProgressBar обычно используют для визуализации процесса выполнения длительной операции.
   3. **Элементами для работы с датами** являются Calendar и DatePicker. Элемент DatePicker позволяет задать дату, набирая её с клавиатуры или применяя выпадающий элемент Calendar.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

*Компоновка* – это процесс размещения визуальных элементов на поверхности родительского элемента. *Контейнер компоновки* – это класс, реализующий определённую логику компоновки дочерних элементов. WPF предлагает ряд стандартных контейнеров, основные контейнеры:

|  |  |
| --- | --- |
| **Имя** | **Описание** |
| **Canvas** | Позволяет элементам позиционироваться по фиксированным координатам |
| **StackPanel** | Размещает элементы в горизонтальный или вертикальный стек. Этот контейнер обычно используется в небольших секциях сложного окна |
| **WrapPanel** | Размещает элементы в сериях строк с переносом. Например, в горизонтальной ориентации WrapPanel располагает элементы в строке слева направо, затем переходит к следующей строке |
| **DockPanel** | Выравнивает элементы по краю контейнера |
| **Grid** | Выстраивает элементы в строки и колонки невидимой таблицы |

Все контейнеры компоновки WPF являются панелями, которые унаследованы от класса System.Windows.Controls.Panel.

**№27. ИСПОЛЬЗОВНИЕ СТИЛЕЙ И ШАБЛОНОВ В WPF.**

*Стиль* ‑ это коллекция значений свойств, которые могут быть применены к элементу. В WPF стили играют ту же роль, которую CSS играет в HTML-разметке. Подобно CSS, стили WPF позволяют определять общий набор характеристик форматирования и применять их по всему приложению для обеспечения согласованности. Стили могут работать автоматически, предназначаться для элементов конкретного типа и каскадироваться через дерево элементов.

Любой стиль – это объект класса System.Windows.Style. Основным свойством стиля является коллекция Setters, в которой каждый элемент задаёт значение для некоторого свойства зависимостей (стили не работают с обычными свойствами – только со свойствами зависимостей). В разметке стиля дочерние элементы с именем <Setter> автоматически помещаются в коллекцию Setters. Свойство стиля TargetType позволяет указать конкретный тип, к которому применяется стиль. В этом случае префикс в установщиках свойств использовать необязательно:

<Style x:Key="baseStyle" TargetType="{x:Type Button}">

<Setter Property="FontSize" Value="22" />

<Setter Property="Background" Value="Purple" />

<Setter Property="Foreground" Value="White" />

</Style>

Определяя стиль, можно построить его на основе стиля-предка. Для этого следует воспользоваться свойством стиля BasedOn. Стили также могут содержать локальные логические ресурсы (коллекция Resources).

*Шаблоны данных* (*data templates*) – механизм для настройки отображения объектов определённого типа. В WPF шаблон данных ‑ это объект класса System.Windows.DataTemplate. Основное свойство шаблона ‑ VisualTree. Оно содержит визуальный элемент, определяющий внешний вид шаблона. Часто этим визуальным элементом является контейнер компоновки. В разметке XAML для задания VisualTree достаточно поместить в DataTemplate дочерний элемент. При формировании VisualTree обычно используется привязка данных для извлечения информации из объекта, для которого применяется шаблон. Сам шаблон данных, как правило, размещают в ресурсах окна или приложения. У класса DataTemplate имеется свойство DataType, определяющее тип данных, к которому будет применяться шаблон.

В WPF списковые элементы управления поддерживают возможность выбора для объекта одного из нескольких шаблонов данных. Выбор шаблона выполняется программно, с помощью создания подкласса для DataTemplateSelector и переопределения метода SelectTemplate().

Затем можно объявить объект TaskTemplateSelector в качестве ресурса и назначить этот ресурс свойству ListBox.ItemTemplateSelector. Объект ListBox вызывает метод SelectTemplate() для каждого элемента в базовой коллекции.

<ListBox Name="lstTasks" Width="250"

HorizontalContentAlignment="Stretch"

ItemTemplateSelector="{StaticResource selector}"/>

При работе с иерархическими элементами управления (например, TreeView) вместо шаблона данных на основе DataTemplate следует использовать HiererchicalDataTemplate. У такого шаблона имеется свойство ItemsSource, которое нужно связать с дочерней коллекцией, и свойство ItemTemplate – дочерний шаблон данных (DataTemplate или HiererchicalDataTemplate).

**№28. ОСНОВНЫЕ КОНЦЕПЦИИ РЕЛЯЦИОННЫХ БД И ЯЗЫКА SQL.**

**Реляционная база данных** – это совокупность отношений, содержащих всю информацию, которая должна храниться в БД. Однако пользователи могут воспринимать такую базу данных как совокупность таблиц.

1. Каждая таблица состоит из однотипных строк и имеет уникальное имя.
2. Строки имеют фиксированное число полей (столбцов) и значений (множественные поля и повторяющиеся группы недопустимы). Иначе говоря, в каждой позиции таблицы на пересечении строки и столбца всегда имеется в точности одно значение или ничего.
3. Строки таблицы обязательно отличаются друг от друга хотя бы единственным значением, что позволяет однозначно идентифицировать любую строку такой таблицы.
4. Столбцам таблицы однозначно присваиваются имена, и в каждом из них размещаются однородные значения данных (даты, фамилии, целые числа или денежные суммы).
5. Полное информационное содержание базы данных представляется в виде явных значений данных и такой метод представления является единственным. В частности, не существует каких-либо специальных "связей" или указателей, соединяющих одну таблицу с другой. Так, связи между строкой с БЛ = 2 таблицы "Блюда" на рис. 3.2 и строкой с ПР = 7 таблицы продукты (для приготовления Харчо нужен Рис), представляется не с помощью указателей, а благодаря существованию в таблице "Состав" строки, в которой номер блюда равен 2, а номер продукта – 7.
6. При выполнении операций с таблицей ее строки и столбцы можно обрабатывать в любом порядке безотносительно к их информационному содержанию. Этому способствует наличие имен таблиц и их столбцов, а также возможность выделения любой их строки или любого набора строк с указанными признаками (например, рейсов с пунктом назначения "Париж" и временем прибытия до 12 часов).

**№29. ТЕХНОЛОГИЯ ADO.NET**

Технология ADO.NET является часть платформы .NET и используется для работы с базами данных. Общая архитектура ADO.NET изображена на рис. 1.



Одним из основных элементов архитектуры ADO.NET является *поставщик* *данных* (*data provider*). Поставщик данных – это совокупность классов, предназначенных для непосредственного взаимодействия с базой. Поставщики данных специфичны для каждого типа СУБД. Унификация поставщиков достигается благодаря тому, что классы любого поставщика реализуют некие стандартные интерфейсы и наследуются от общих базовых классов. Эти базовые элементы содержатся в пространстве имён System.Data.Common.

Любой поставщик данных содержит четыре основных класса, унаследованных от DbConnection, DbCommand, DbDataReader и DbDataAdapter. Имена классов специфичны у каждого поставщика, но обычно различаются только префиксы имён. Так, поставщик данных для MS SQL Server содержит классы SqlConnection, SqlCommand, SqlDataReader, SqlDataAdapter. Назначение класса DbConnection – установка и поддержка соединения с базой данных. Класс DbCommand служит для выполнения запросов и команд к базе. Можно выполнить команды, не возвращающие данных (например, создать таблицу), и запросы, возвращающие скалярное значение или набор данных. В последнем случае для чтения данных используется класс DbDataReader – *ридер*. Отличительной особенностью ридера является то, что он представляет собой однонаправленный курсор данных в режиме «только-для-чтения». Класс DbDataAdapter служит своеобразным «мостом» между поставщиком данных и рассоединенным набором данных. Этот класс содержит четыре команды для выборки, обновления, вставки и удаления данных.

В состав поставщика данных входят некоторые вспомогательные элементы. Класс DbConnectionStringBuilder используется для построения строк соединения с базой. Класс DbTransaction служит для описания транзакции данных. Класс DbParameter описывает параметры команды к БД.

Вторым элементом ADO.NET является набор классов, представляющих рассоединенный набор данных. Эти классы универсальны и не зависят от поставщика данных. Главным компонентом набора является класс DataSet, агрегирующий объекты остальных классов. Класс DataTable служит для описания таблиц базы. Элементами класса DataTable являются коллекции объектов DataColumn (колонки таблицы), DataRow (строки таблицы) и Constraint (ограничения на значения элементов таблицы). Класс DataRelation описывает связи между таблицами. В ADO.NET реализована модель получения поставщиков данных, основанная на шаблоне проектирования «фабрика». В пространстве имён System.Data.Common имеется DbProviderFactory, который возвращает конкретный тип поставщика по указанному строковому имени, например "System.Data.SqlClient". Использование абстрактных базовых классов облегчает процесс создания поставщиков и позволяет управлять соединениями и командами в режиме исполнения.

**№30. ОБЪЕКТНО-РЕЛЯЦИОННОЕ ОТОБРАЖЕНИЕ.**

Технология программирования, которая связывает БД с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

Суть задачи состоит в преобразовании объектов в форму, в которой они могут быть сохранены в файлах или БД, и которые легко могут быть извлечены в последующем, с сохранением свойств объектов и отношений между ними. Эти объекты называют «хранимыми» (англ. persistent).

Решение проблемы хранения данных существует — это реляционные системы управления БД. Использование реляционной БД для хранения объектно-ориентированных данных приводит к семантическому провалу, заставляя программистов писать программное обеспечение, которое должно уметь как обрабатывать данные в объектно-ориентированном виде, так и уметь сохранить эти данные в реляционной форме. Эта постоянная необходимость в преобразовании между двумя разными формами данных не только сильно снижает производительность, но и создает трудности для программистов, так как обе формы данных накладывают ограничения друг на друга.

Реляционные базы данных используют набор таблиц, представляющих простые данные. Дополнительная или связанная информация хранится в других таблицах. Часто для хранения одного объекта в реляционной базе данных используется несколько таблиц; это, в свою очередь, требует применения операции JOIN для получения всей информации, относящейся к объекту, для ее обработки. Например, в рассмотренном варианте с записной книгой, для хранения данных, скорее всего, будут использоваться как минимум две таблицы: люди и адреса, и, возможно, даже таблица с телефонными номерами.

Так как системы управления реляционными базами данных обычно не реализуют реляционного представления физического уровня связей, выполнение нескольких последовательных запросов может быть слишком затратно. В частности, один запрос вида «найти такого-то пользователя и все его телефоны и все его адреса и вернуть их в таком формате», скорее всего, будет выполнен быстрее серии запросов вида «Найти пользователя. Найти его адреса. Найти его телефоны». Это происходит благодаря работе оптимизатора и затратам на синтаксический анализ запроса.

Некоторые реализации ORM автоматически синхронизируют загруженные в память объекты с базой данных. Для того чтобы это было возможным, после создания объект-в-SQL-преобразующего SQL-запроса полученные данные копируются в поля объекта, как во всех других реализациях ORM. После этого объект должен следить за изменениями этих значений и записывать их в базу данных.

Системы управления реляционными базами данных показывают хорошую производительность на глобальных запросах, которые затрагивают большой участок базы данных, но объектно-ориентированный доступ более эффективен при работе с малыми объёмами данных, так как это позволяет сократить семантический провал между объектной и реляционной формами данных.

При одновременном существовании этих двух разных миров увеличивается сложность объектного кода для работы с реляционными базами данных, и он становится более подвержен ошибкам. Разработчики программного обеспечения, основывающегося на базах данных, искали более легкий способ достижения постоянства их объектов.

Некоторые пакеты решают эту проблему, предоставляя библиотеки классов, способных выполнять такие преобразования автоматически. Имея список таблиц в базе данных и объектов в программе, они автоматически преобразуют запросы из одного вида в другой. В результате запроса объекта «человек» (из примера с адресной книгой) необходимый SQL-запрос будет сформирован и выполнен, а результаты «волшебным» образом преобразованы в объекты «номер телефона» внутри программы.

Но ORM избавляет программиста от написания большого количества кода, часто однообразного и подверженного ошибкам, тем самым значительно повышая скорость разработки. Кроме того, большинство современных реализаций ORM позволяют программисту при необходимости самому жёстко задать код SQL-запросов, который будет использоваться при тех или иных действиях (сохранение в базу данных, загрузка, поиск и т. д.) с постоянным объектом.