Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ |
| Заместитель директора  по учебной работе |
| \_\_\_\_\_\_\_\_\_ И.В.Малафей |
| «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ |

**КОНСТРУИРОВАНИЕ ПРОГРАММ**

**И ЯЗЫКИ ПРОГРАММИРОВАНИЯ**

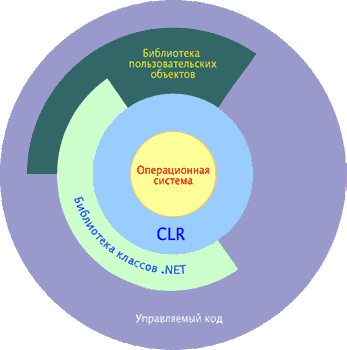
**Вопросы к экзамену для учащихся 3 курса дневной формы получения образования специальности 2-40 01 01 «Программное обеспечение информационных технологий»**

Составлены на основании учебной программы по учебному предмету «Конструирование программ и языки программирования», утвержденной директором «Колледжа Бизнеса и права» 06.07.2021

1. Опишите основные этапы обработки программы на основе платформы .NET. Опишите принцип компиляции программ под управлением среды CLR с использованием JIT-компилятора. Напишите код заготовки консольной программы и прокомментируйте все ее части.
2. Написание исходного кода: На этом этапе программист создает исходный код на языке программирования, совместимом с платформой .NET, таком как C# или VB.NET. Исходный код содержит инструкции, которые определяют поведение программы.
3. Компиляция: На этом этапе исходный код компилируется в промежуточный язык, называемый промежуточным представлением (Intermediate Language, IL), с помощью компилятора языка .NET, такого как C# Compiler (csc.exe). IL является независимым от аппаратного обеспечения и платформы представлением программы.
4. JIT-компиляция: Когда программа запускается, CLR (Common Language Runtime) загружает IL-код и передает его JIT-компилятору (Just-In-Time Compiler). JIT-компилятор преобразует IL-код в машинный код, специфичный для конкретной аппаратной платформы, на которой выполняется программа. Это происходит на этапе выполнения программы. JIT-компиляция позволяет оптимизировать выполнение программы и улучшить производительность, так как машинный код выполняется непосредственно процессором.
5. Выполнение программы: После JIT-компиляции машинный код выполняется CLR. CLR предоставляет различные службы, такие как управление памятью, сборка мусора и обработка исключений. Он также обеспечивает доступ к .NET Framework и другим библиотекам, которые программист может использовать для разработки приложений.
6. Опишите каркас платформы .NET. Нарисуйте структуру платформы .NET. Перечислите основные составляющие алфавита языка С#.

Платформа .NET (от англ. "Network Enabled Technologies") является программной платформой, разработанной компанией Microsoft. Она предоставляет среду выполнения для разработки, развертывания и выполнения приложений, независимо от аппаратного обеспечения и операционной системы. Вот каркас платформы .NET:

1. Компиляторы: Платформа .NET включает компиляторы для различных языков программирования, таких как C#, VB.NET, F# и других. Компиляторы преобразуют исходный код на этих языках в промежуточное представление (IL), которое затем может быть выполнено CLR.
2. CLR (Common Language Runtime): Это основная часть платформы .NET. CLR предоставляет среду выполнения, которая управляет выполнением программы на основе .NET. Он обеспечивает функции, такие как управление памятью, сборка мусора, обработка исключений и безопасность типов. CLR также содержит JIT-компилятор, который преобразует промежуточное представление (IL) в машинный код, который может выполняться на конкретной платформе.
3. Библиотеки классов: Платформа .NET включает набор библиотек классов, которые предоставляют различные функциональные возможности для разработки приложений. Эти библиотеки предоставляют классы и методы для работы с файлами, сетью, базами данных, графикой, XML, асинхронным программированием и многим другим.
4. Языки программирования: Платформа .NET поддерживает различные языки программирования, включая C#, VB.NET, F#, C++/CLI и другие. Они имеют доступ к общей инфраструктуре и могут использовать библиотеки классов .NET.
5. Инструменты разработки: Для разработки приложений на платформе .NET предоставляются различные инструменты, такие как интегрированная среда разработки (IDE) Visual Studio, отладчики, профилировщики и другие утилиты.



+----------------------+

| .NET Application |

+----------------------+

| CIL Executable |

+----------------------+

| Common Language |

| Runtime |

+----------------------+

| Base Class Library |

+----------------------+

| Language-Specific|

| Libraries |

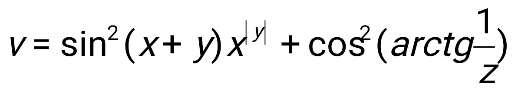
+----------------------+

| Language-Specific|

| Compilers |

+----------------------+

1. Опишите каждую составляющую алфавита языка С#. Дайте определение понятию лексем в языке С#. Напишите фрагмент кода программы для расчета значения по заданной формуле

******

Составляющие алфавита языка C# включают:

1. Буквы: Язык C# использует латинский алфавит для идентификаторов, ключевых слов и других элементов кода.
2. Цифры: Десятичные цифры от 0 до 9 используются для представления чисел и констант.
3. Знаки пунктуации: Включают знаки препинания, такие как точка, запятая, двоеточие, точка с запятой и другие, используемые для разделения элементов кода.
4. Операторы: Язык C# содержит операторы для выполнения различных операций, такие как арифметические операторы (+, -, \*, /), операторы сравнения (==, !=, >, <), операторы присваивания (=), логические операторы (&&, ||, !) и многое другое.
5. Пробелы: Пробелы используются для разделения элементов кода и обеспечения читаемости.
6. Комментарии: Комментарии используются для добавления пояснений к коду и игнорируются компилятором при выполнении программы. В C# есть однострочные комментарии (//) и многострочные комментарии (/\* \*/).

Лексема в языке C# представляет собой минимальную синтаксическую единицу программы. Лексемы могут быть идентификаторами, ключевыми словами, операторами, разделителями и т.д. Компилятор C# анализирует и разбирает исходный код программы на лексемы, чтобы определить их семантику и выполнять соответствующие действия.

Пример фрагмента кода программы для расчета значения по заданной формуле:

csharp

using System;

class Program

{

static void Main()

{

double x = 1.5;

double y = 2.3;

double z = 4.7;

double v = Math.Pow(Math.Sin(x + y), 2) \* Math.Pow(x, Math.Abs(y)) + Math.Pow(Math.Cos(Math.Atan(1 / z)), 2);

Console.WriteLine("Результат: " + v);

}

}

В этом фрагменте кода используется класс Math из пространства имен System для выполнения математических операций. Значения переменных x, y и z заданы исходными значениями. Формула v = sin^2(x+y) \* x^|y| + cos^2(arctg(1/z)) вычисляется и выводится на консоль.

1. Дайте определение понятию переменной. Объясните разницу между объявлением и инициализацией переменной. Опишите особенности названия переменных в С#. Приведите примеры.

Переменная в программировании является именованной областью памяти, предназначенной для хранения данных. Она имеет тип данных, который определяет, какие значения могут быть сохранены в переменной, и может изменяться в процессе выполнения программы.

Разница между объявлением и инициализацией переменной:

* Объявление переменной: Это шаг, при котором программист указывает компилятору тип данных переменной и ее имя. Объявление создает переменную, но не присваивает ей конкретное значение. На этом этапе переменная содержит некоторое значение по умолчанию в зависимости от ее типа данных.

Пример объявления переменной в C#:

int age; // Объявление переменной "age" типа int

* Инициализация переменной: Это шаг, при котором программист присваивает переменной конкретное значение. Инициализация может происходить одновременно с объявлением или в любой момент позже в программе.

Пример инициализации переменной в C#:

int age; // Объявление переменной "age" типа int

age = 25; // Инициализация переменной "age" значением 25

Особенности названия переменных в C#:

1. Имена переменных могут состоять из букв, цифр и символа подчеркивания (\_).
2. Имена переменных должны начинаться с буквы или символа подчеркивания (\_). Они не могут начинаться с цифры.
3. Имена переменных чувствительны к регистру, то есть "age" и "Age" будут считаться разными переменными.
4. Имена переменных не могут быть ключевыми словами языка C#, такими как "int", "string" и т.д.

Примеры правильных имен переменных в C#:

int age; // Переменная для хранения возраста

string name; // Переменная для хранения имени

float temperature; // Переменная для хранения температуры

bool isAlive; // Переменная для хранения состояния жизни

Важно выбирать информативные и понятные имена переменных, которые отражают их предназначение и помогают лучше понимать код программы.

1. Поясните что определяет тип данных. Что такое var? Перечислите типы данных в C# с указанием системного типа.

Тип данных в программировании определяет набор значений, которые может принимать переменная, а также операции, которые можно выполнять над этими значениями. Тип данных определяет, как компьютер должен интерпретировать и обрабатывать биты памяти, выделенные для переменной.

Ключевое слово var в C# используется для объявления переменной, когда тип данных может быть определен компилятором на основе выражения, к которому переменная присваивается. Компилятор автоматически определяет тип переменной на основе типа выражения справа от оператора присваивания.

Пример использования var:

var age = 25; // Компилятор определит тип данных переменной "age" как int

var name = "John"; // Компилятор определит тип данных переменной "name" как string

Типы данных в C# с указанием системного типа (System Type):

1. Целочисленные типы:
   * int (System.Int32) - 32-битное целое число.
   * long (System.Int64) - 64-битное целое число.
   * short (System.Int16) - 16-битное целое число.
   * byte (System.Byte) - 8-битное беззнаковое целое число.
2. Вещественные типы:
   * float (System.Single) - 32-битное число с плавающей запятой.
   * double (System.Double) - 64-битное число с плавающей запятой.
3. Символьный тип:
   * char (System.Char) - 16-битный символ Unicode.
4. Логический тип:
   * bool (System.Boolean) - логическое значение true или false.
5. Строковый тип:
   * string (System.String) - последовательность символов.
6. Другие типы данных:
   * decimal (System.Decimal) - десятичное число с фиксированной точностью.
   * DateTime (System.DateTime) - тип, представляющий дату и время.
   * object (System.Object) - базовый тип для всех остальных типов данных.

В C# также есть возможность определять пользовательские типы данных с помощью классов, структур и перечислений.

1. Опишите процесс явного и неявного преобразования типов в языке C#. Перечислите все безопасные преобразования типов в С#. Для переменных int b = 300; long a; byte d; осуществите преобразование типа с потерей и без потери данных.

В C# есть два способа преобразования типов: явное и неявное.

1. Неявное преобразование типов:
   * Неявное преобразование типов происходит автоматически и без явного указания программистом.
   * Оно происходит, когда преобразование не приводит к потере данных или урезанию точности.
   * Обычно неявное преобразование происходит между совместимыми типами данных, где целевой тип имеет больший диапазон или более точность, чем исходный тип.

Примеры неявного преобразования типов:

int a = 5;

long b = a; // Неявное преобразование int в long

float c = 3.14f;

double d = c; // Неявное преобразование float в double

1. Явное преобразование типов:
   * Явное преобразование типов требует явного указания программистом и может привести к потере данных или урезанию точности.
   * Оно используется, когда преобразование между типами неявно невозможно или может вызвать потерю данных.
   * Явное преобразование выполняется с использованием оператора приведения типа (тип) выражение.

Примеры явного преобразования типов:

double a = 3.14;

int b = (int)a; // Явное преобразование double в int

int c = 255;

byte d = (byte)c; // Явное преобразование int в byte

Безопасные преобразования типов в C# включают:

1. Преобразование между числовыми типами, когда целевой тип имеет больший размер или точность.
2. Преобразование к более общим типам данных, таким как преобразование производных классов к базовым классам.
3. Преобразование между перечислениями и их базовыми типами (например, int).
4. Преобразование между строками и числовыми типами, когда строка содержит корректное числовое представление.

Пример преобразования с потерей данных:

int b = 300;

byte d = (byte)b; // Явное преобразование int в byte с потерей данных

Console.WriteLine(d); // Вывод: 44 (300 % 256 = 44)

Пример преобразования без потери данных:

int b = 300;

long a = b; // Неявное преобразование int в long без потери данных

Console.WriteLine(a); // Вывод: 300

Важно помнить, что при явном преобразовании типов может произойти потеря данных или изменение значения, поэтому нужно быть осторожным при использовании таких преобразований и убедиться, что они не приведут к ошибкам или нежелательным результатам.

1. Классифицируйте по группам операции языка C#. Опишите принцип работы тернарной условной операции. С помощью тернарного оператора найдите максимальное из двух целых чисел.

Операции в языке C# можно классифицировать следующим образом:

1. Арифметические операции:
   * Сложение (+): складывает два операнда.
   * Вычитание (-): вычитает один операнд из другого.
   * Умножение (\*): умножает два операнда.
   * Деление (/): делит один операнд на другой.
   * Остаток от деления (%): возвращает остаток от деления одного операнда на другой.
2. Операции присваивания:
   * Присваивание (=): присваивает значение правого операнда левому операнду.
   * Комбинированные операции присваивания (+=, -=, \*=, /=, %= и другие): выполняют операцию и присваивание в одном выражении.
3. Операции сравнения:
   * Равенство (==): возвращает true, если два операнда равны.
   * Неравенство (!=): возвращает true, если два операнда не равны.
   * Больше (>), меньше (<), больше или равно (>=), меньше или равно (<=): сравнивают два операнда.
4. Логические операции:
   * Логическое И (&&): возвращает true, если оба операнда являются true.
   * Логическое ИЛИ (||): возвращает true, если хотя бы один операнд является true.
   * Логическое НЕ (!): инвертирует значение операнда.
5. Побитовые операции:
   * Побитовое И (&): выполняет побитовую конъюнкцию двух операндов.
   * Побитовое ИЛИ (|): выполняет побитовую дизъюнкцию двух операндов.
   * Побитовое исключающее ИЛИ (^): выполняет побитовое исключающее ИЛИ двух операндов.
   * Побитовый сдвиг влево (<<): сдвигает биты левого операнда влево на количество позиций, указанных правым операндом.
   * Побитовый сдвиг вправо (>>): сдвигает биты левого операнда вправо на количество позиций, указанных правым операндом.
6. Тернарная условная операция:
   * Тернарная условная операция (condition ? expression1 : expression2) позволяет выполнить выбор между двумя выражениями на основе условия.
   * Если условие истинно, возвращается значение expression1, в противном случае возвращается значение expression2.

Принцип работы тернарной условной операции:

int max = (a > b) ? a : b;

В приведенном примере, если a больше b, то значение a будет присвоено переменной max, иначе значение b будет присвоено переменной max. Таким образом, результатом будет максимальное из двух чисел.

Пример использования тернарного оператора для нахождения максимального из двух целых чисел:

int a = 5;

int b = 7;

int max = (a > b) ? a : b;

Console.WriteLine(max); // Вывод: 7

В данном случае, так как b (значение 7) больше a (значение 5), то переменной max будет присвоено значение b (7), которое является максимальным из двух чисел.

1. Перечислите операции, относящиеся к арифметическим. Приведите примеры операций присваивания. Поясните в чем заключается разница между унарными операциями: (++а)-5 и (а++)-5. Вычислите результат при а=10.

Операции, относящиеся к арифметическим, включают:

1. Сложение (+): складывает два операнда.
2. Вычитание (-): вычитает один операнд из другого.
3. Умножение (\*): умножает два операнда.
4. Деление (/): делит один операнд на другой.
5. Остаток от деления (%): возвращает остаток от деления одного операнда на другой.

Примеры операций присваивания:

1. Присваивание (=): присваивает значение правого операнда левому операнду.

· int a = 5;

int b;

b = a; // Присваивание значения переменной "a" переменной "b"

· Комбинированные операции присваивания (+=, -=, \*=, /=, %= и другие): выполняют операцию и присваивание в одном выражении.

int a = 10;

a += 5; // Идентично a = a + 5, значение переменной "a" станет 15

Разница между унарными операциями ++а - 5 и а++ - 5 заключается в порядке выполнения операций и результате:

1. ++а - 5:
   * Унарная операция инкремента ++а будет выполнена первой. Значение переменной а увеличится на 1.
   * Затем вычитается 5 из полученного значения переменной а.
   * Итоговый результат - значение а после увеличения на 1 минус 5.
2. а++ - 5:
   * Значение переменной а вычитается 5.
   * Затем унарная операция инкремента а++ выполняется, увеличивая значение а на 1, но это увеличение не влияет на итоговый результат.
   * Итоговый результат - значение а до увеличения на 1 минус 5.

Вычисление результата при а = 10:

* Для выражения ++а - 5:
  + Инкремент ++а увеличит а до 11.
  + Итоговый результат: 11 - 5 = 6.
* Для выражения а++ - 5:
  + Итоговый результат: 10 - 5 = 5.
  + Значение переменной а будет увеличено на 1 после выполнения этого выражения.

Таким образом, при а = 10 результат для выражения ++а - 5 будет 6, а для выражения а++ - 5 будет 5, а значение переменной а после этого будет равно 11.

1. Перечислите все унарные операции языка C# и приведите примеры. Перечислите битовые операции. Объясните по какому принципу заполняются освобождающиеся позиции при выполнении операций сдвига. Опишите на примере результат выполнения операций &, | , ^ для выражения: а=b…c, если b=1010, c= 0110.

В языке C# унарные операции могут быть применены к одному операнду. Вот некоторые примеры унарных операций в C#:

1. Унарный плюс (+): Используется для явного указания положительного значения переменной. Пример: int a = +5;
2. Унарный минус (-): Используется для изменения знака переменной на противоположный. Пример: int b = -10;
3. Инкремент (++): Увеличивает значение переменной на 1. Пример: int c = 5; c++;
4. Декремент (--): Уменьшает значение переменной на 1. Пример: int d = 8; d--;
5. Логическое отрицание (!): Инвертирует булево значение (true становится false и наоборот). Пример: bool flag = true; bool invertedFlag = !flag;

Битовые операции в языке C#:

1. Побитовое И (&): выполняет побитовую конъюнкцию двух операндов. Например: int result = b & c; // result будет равно 0000
2. Побитовое ИЛИ (|): выполняет побитовую дизъюнкцию двух операндов. Например: int result = b | c; // result будет равно 1110
3. Побитовое исключающее ИЛИ (^): выполняет побитовое исключающее ИЛИ двух операндов. Например: int result = b ^ c; // result будет равно 1110

a = b & c = 1010

a = b | c = 0010

a = b ^ c = 1110

1. Поясните синтаксис и особенности использования условного оператора if в С#. С помощью условного оператора написать программу для расчета корней квадратного уравнения.

if(условие)

{

выполняемые инструкции

}

или

if(условие)

{

выполняемые инструкции

}

Else

{

выполняемые инструкции

}

1. Поясните синтаксис и особенности использования оператора выбора switch-case-default в С#. Приведите фрагмент программы, позволяющей узнать пору года при вводе числа (месяца) от 1 до 12 (1 – зима, 2 – зима, 3 – весна…). В случае ввода не верных значений выбрасывать ошибку.

switch (выражение)

{

case значение1:

код,выполняемый если выражение имеет значение1

break;

case значение2:

код,выполняемый если выражение имеет значение1

break;

//.............

case значениеN:

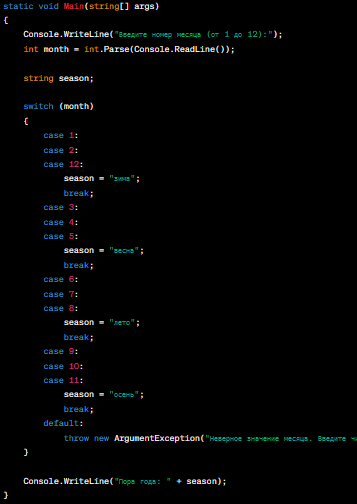
код, выполняемый если выражение имеет значениеN

break;

default:

код, выполняемый если выражение не имеет ни одно из выше указанных значений

break;

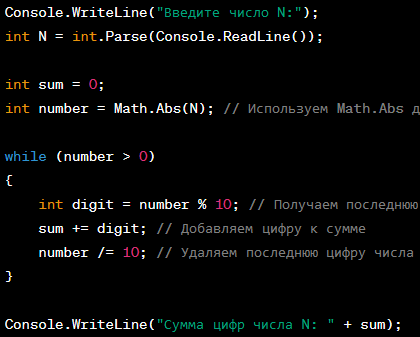


1. Поясните синтаксис и особенности использования операторов цикла while и do-while в С#. Опишите разницу между оператором цикла с пред- и постусловием. С помощью операторов цикла while или do-while написать программу для суммы цифр числа N. Число N может иметь любое количество знаков.

Оператор цикла while предназначен для организации циклического процесса, в котором выполнение каждой следующей итерации определяется на основе истинности некоторого условия. Оператор цикла while еще называют оператором цикла с предусловием.

Оператор цикла с предусловием (while) проверяет условие перед выполнением каждой итерации цикла. Если условие истинно, то выполняется код внутри цикла, и после выполнения кода снова происходит проверка условия. Если условие ложно, то цикл завершается и программа продолжает выполнение после цикла.

Оператор цикла с постусловием (do-while) проверяет условие после выполнения кода внутри цикла. В этом случае код внутри цикла всегда выполняется хотя бы один раз, даже если условие изначально ложно. После выполнения кода внутри цикла происходит проверка условия. Если условие истинно, то цикл продолжается и код внутри цикла выполняется снова. Если условие ложно, то цикл завершается и программа продолжает выполнение после цикла.



1. Поясните синтаксис и особенности использования операторов цикла for и foreach в С#. Опишите разницу между циклами for и foreach. С помощью операторов цикла for или foreach написать программу для суммы всех четных двузначных чисел.

Операторы цикла for и foreach в C# предоставляют возможность выполнения итераций (повторений) некоторого блока кода. Они имеют разный синтаксис и применяются в различных ситуациях.

for (инициализация; условие; итерация)

{

// Код, выполняемый внутри цикла

}

foreach (тип\_элемента элемент in коллекция)

{

// Код, выполняемый внутри цикла

}

Код программы по нахождению суммы двухначных чисел

int sum = 0;

for (int i = 10; i <= 99; i += 2)

{

sum += i;

}

1. Для чего нужны операторы передачи управления? Поясните особенности использования операторов continue, break, go-to и return в С#. С помощью операторов управления найдите в целочисленном массиве сумму между первым положительным и последним нулевым элементами.

Операторы передачи управления в языке C# позволяют изменять поток выполнения программы, переходить к другим частям кода или прекращать выполнение определенных блоков кода. Они предоставляют возможность более гибкого управления исполнением программы в различных ситуациях.

Оператор continue:

* + Используется внутри цикла.
  + Прерывает текущую итерацию цикла и переходит к следующей итерации.
  + Оставшийся код внутри цикла после оператора continue не выполняется.

Оператор break:

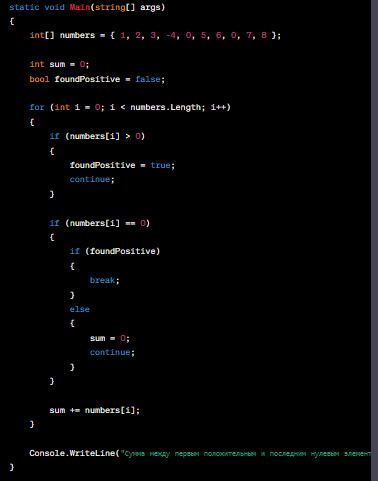
* + Используется внутри цикла или switch-конструкции.
  + Прерывает выполнение текущего цикла или switch-конструкции и выходит из них.
  + После оператора break выполнение программы продолжается с первого оператора после цикла или switch-конструкции.

Оператор go-to:

* + Позволяет осуществить безусловный переход к указанной метке в коде.
  + Использование оператора go-to не рекомендуется, так как оно может сделать код менее читаемым и усложнить его понимание.

Оператор return:

* + Используется внутри метода или функции.
  + Прекращает выполнение метода и возвращает указанное значение (если метод возвращает значение) или просто завершает выполнение метода (если метод не возвращает значение).



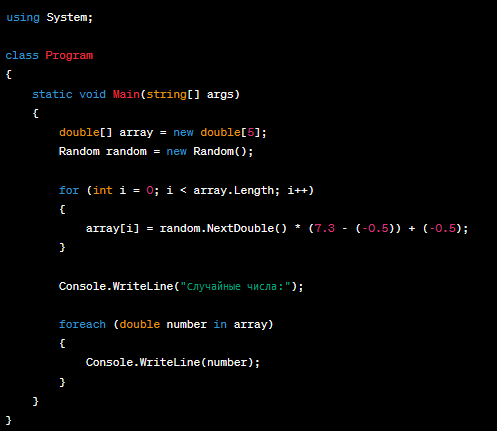
1. Опишите назначение методов класса Random. Опишите синтаксис получения псевдослучайной последовательности целых и дробных чисел с изменением и без изменения базы генерации. Напишите фрагмент программы заполнения double-массива из пяти элементов случайными целыми числами в диапазоне от -0,5 до 7,3.

Методы класса Random в языке C# используются для генерации псевдослучайных чисел. Класс Random предоставляет различные методы для генерации случайных чисел разных типов и в разных диапазонах.

Назначение методов класса Random:

1. Next(): Генерирует случайное 32-битное целое число в диапазоне от 0 до Int32.MaxValue.
2. Next(minValue, maxValue): Генерирует случайное 32-битное целое число в указанном диапазоне от minValue до maxValue (исключая maxValue).
3. NextDouble(): Генерирует случайное число с плавающей точкой типа double в диапазоне от 0,0 до 1,0.
4. NextBytes(byte[] buffer): Заполняет указанный массив buffer случайными байтами.

Синтаксис получения псевдослучайной последовательности целых и дробных чисел с изменением и без изменения базы генерации:



1. Дайте определение понятию массив. Перечислите способы объявления одномерного массива. Напишите код для инициализации одномерного целочисленного массива значениями с консоли. Составьте программу, которая найдет в одномерном массиве минимальной положительный элемент. Значение найденного элемента выведите на консоль.

Массив представляет набор однотипных данных.

int[] numbers;

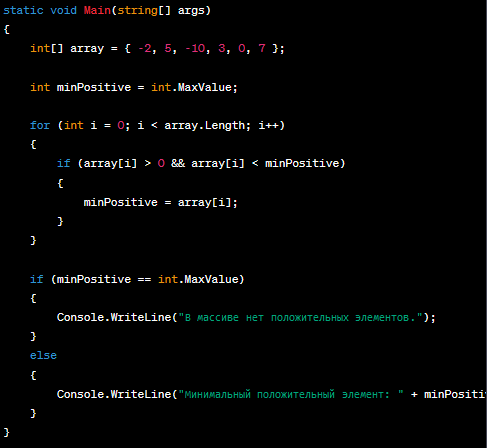
int[] nums = new int[4];

int[] nums2 = new int[4] { 1, 2, 3, 5 };

int[] nums3 = new int[] { 1, 2, 3, 5 };

int[] nums4 = new[] { 1, 2, 3, 5 };

int[] nums5 = { 1, 2, 3, 5 };



1. Дайте определение понятию длина массива и индекс элемента массива. Перечислите способы объявления двумерного массива. Напишите код для инициализации двумерного целочисленного массива значениями с консоли. Составьте программу, которая найдет в двумерном массиве сумму элементов на главной диагонали. Значение суммы выведите на консоль.

Длина массива (array length) - это количество элементов, содержащихся в массиве.

Индекс элемента массива (array index) - это числовой указатель на определенный элемент массива.

int[,] nums1;

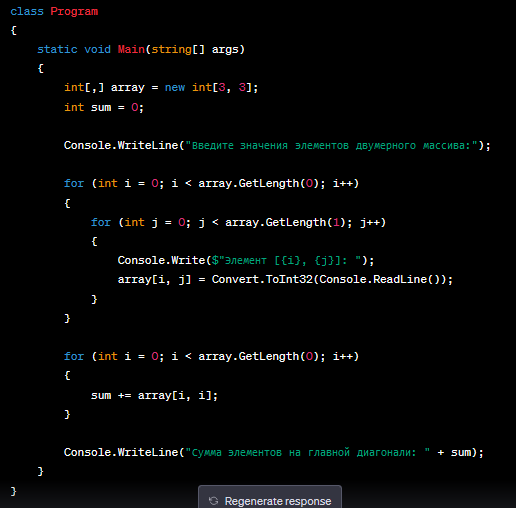
int[,] nums2 = new int[2, 3];

int[,] nums3 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };

int[,] nums4 = new int[,] { { 0, 1, 2 }, { 3, 4, 5 } };

int[,] nums5 = new [,]{ { 0, 1, 2 }, { 3, 4, 5 } };

int[,] nums6 = { { 0, 1, 2 }, { 3, 4, 5 } };



18. Опишите назначение класса Array. Приведите примеры его методов и/или свойств и укажите их назначение. Составьте программу с использованием методов класса Array, которая найдет в одномерном массиве сумму элементов, расположенных между первым и последним нулевым элементами. Значение суммы выведите на консоль.

Класс Array используется для работы с одномерными массивами. Некоторые его методы и свойства:

* Length - возвращает длину массива.
* GetValue(index) - возвращает значение элемента массива по индексу.
* SetValue(value, index) - устанавливает значение элемента массива по индексу.
* IndexOf(value) - возвращает индекс первого вхождения указанного значения.

Программа, которая находит сумму элементов между первым и последним нулевыми элементами одномерного массива, и выводит эту сумму на консоль, может выглядеть так:

using System;

class Program

{

static void Main()

{

int[] array = { 1, 2, 0, 3, 4, 0, 5, 6 };

int firstZeroIndex = Array.IndexOf(array, 0);

int lastZeroIndex = Array.LastIndexOf(array, 0);

int sum = 0;

for (int i = firstZeroIndex + 1; i < lastZeroIndex; i++)

{

sum += array[i];

}

Console.WriteLine("Сумма элементов между первым и последним нулевыми элементами: " + sum);

}

}

В данном примере массив array содержит числа, алгоритм ищет первое и последнее вхождение нуля, а затем находит сумму элементов, находящихся между ними. Результат выводится на консоль.

19. Дайте определение понятию «метод». Опишите разницу между функцией и методом. Укажите назначение параметра static для методов. Опишите статический метод int SumOfDigits(int n), который принимает некоторое целое число (его длина может быть любой) и возвращает сумму его цифр. Продемонстрируйте в Main вызов и результат работы этого метода. Продемонстрировать вызов метода в Main.

Метод - это подпрограмма или функция, которая выполняет определенное действие или решает определенную задачу. Он связан с определенным объектом или классом и может вызываться для выполнения операций с этим объектом или классом.

Разница между функцией и методом состоит в том, что функция является независимой от объектов подпрограммой, в то время как метод привязан к определенному объекту или классу и может иметь доступ к его свойствам и методам.

Параметр static для методов означает, что метод принадлежит классу, а не конкретному экземпляру объекта. Такие методы могут быть вызваны без создания экземпляра класса и работают на уровне класса, а не на уровне конкретного объекта.

Пример статического метода SumOfDigits, который принимает целое число и возвращает сумму его цифр:

using System;

class Program

{

static int SumOfDigits(int n)

{

int sum = 0;

while (n != 0)

{

sum += n % 10;

n /= 10;

}

return sum;

}

static void Main()

{

int number = 12345;

int sum = SumOfDigits(number);

Console.WriteLine("Сумма цифр числа {0} равна {1}", number, sum);

}

}

В данном примере метод SumOfDigits принимает целое число n и вычисляет сумму его цифр, используя операцию деления на 10 и остаток от деления. В методе Main вызывается этот метод для числа 12345, и результат выводится на консоль.

20. Перечислите способы передачи параметров в методах. Укажите разницу между передачей параметров по ссылке и по значению. Поясните разницу между передачей ref- и out-параметра в метод. Опишите статический метод, который меняет первый и последний элемент в массиве. Продемонстрируйте в Main вызов и результат работы этого метода.

Способы передачи параметров в методах:

1. Передача параметра по значению: Значение параметра копируется в метод, и изменения в методе не влияют на оригинальное значение.
2. Передача параметра по ссылке: Ссылка на оригинальный объект передается в метод, и изменения в методе могут изменить оригинальное значение.

Разница между передачей параметров по ссылке и по значению заключается в том, что при передаче по значению изменения в методе не влияют на оригинальное значение, а при передаче по ссылке изменения в методе могут изменить оригинальное значение.

Разница между ref-параметром и out-параметром:

* ref-параметр: Требует, чтобы переменная, переданная в качестве аргумента, была инициализирована до передачи в метод. Может быть использован для передачи значения и получения измененного значения обратно в вызывающий код.
* out-параметр: Позволяет методу вернуть несколько значений. Не требует, чтобы переменная была инициализирована до передачи в метод. Метод должен инициализировать переменную внутри себя перед выходом из метода.

Пример статического метода SwapFirstAndLast, который меняет первый и последний элемент в массиве:

using System;

class Program

{

static void SwapFirstAndLast(int[] array)

{

if (array.Length >= 2)

{

int temp = array[0];

array[0] = array[array.Length - 1];

array[array.Length - 1] = temp;

}

}

static void Main()

{

int[] numbers = { 1, 2, 3, 4, 5 };

Console.WriteLine("Исходный массив: " + string.Join(", ", numbers));

SwapFirstAndLast(numbers);

Console.WriteLine("Массив после замены первого и последнего элементов: " + string.Join(", ", numbers));

}

}

21. Перечислите особенности работы со строками в классе String. Приведите примеры его методов и/или свойств и укажите их назначение. Составьте программу, которая посчитает количество цифр в заданной строке (string). Найденное количество выведите на консоль.

Особенности работы со строками в классе String в C#:

* Строки в классе String являются неизменяемыми (immutable), то есть их значения не могут быть изменены после создания.
* Класс String предоставляет множество методов для манипуляции и работы со строками.

Примеры методов и свойств класса String в C#:

1. Length - свойство, которое возвращает количество символов в строке.
2. Substring(startIndex, length) - метод, который возвращает подстроку, начиная с указанного индекса и заданной длины.
3. IndexOf(str) - метод, который возвращает индекс первого вхождения подстроки str в строку.
4. Split(separator) - метод, который разделяет строку на подстроки, используя указанный разделитель.
5. ToLower() - метод, который возвращает новую строку, в которой все символы приведены к нижнему регистру.

Программа на C# для подсчета количества цифр в заданной строке:

using System;

class Program

{

static void Main(string[] args)

{

string input = "abc123xyz456";

int count = 0;

foreach (char c in input)

{

if (char.IsDigit(c))

{

count++;

}

}

Console.WriteLine("Количество цифр: " + count);

}

}

22. Перечислите особенности работы со строками в классе StringBuilder. Приведите примеры его методов и/или свойств и укажите их назначение. Составьте программу, которая добавляет к исходной строке (StringBuilder) слово «Hello». Измененную строку выведите на консоль.

Особенности работы со строками в классе StringBuilder в C#:

* Строки в классе StringBuilder являются изменяемыми (mutable), то есть их значения могут быть изменены после создания без создания нового экземпляра.
* Класс StringBuilder предоставляет методы для манипуляции и работы со строками, особенно при выполнении операций изменения и конкатенации.

Примеры методов и свойств класса StringBuilder в C#:

1. Append(str) - метод, который добавляет указанную строку в конец текущей строки.
2. Insert(index, str) - метод, который вставляет указанную строку по указанному индексу в текущую строку.
3. Remove(startIndex, length) - метод, который удаляет указанное количество символов начиная с указанного индекса в текущей строке.
4. ToString() - метод, который возвращает текущую строку в виде обычной строки типа string.

Программа на C# для добавления слова "Hello" к исходной строке StringBuilder и вывода измененной строки на консоль:

using System;

using System.Text;

class Program

{

static void Main(string[] args)

{

StringBuilder sb = new StringBuilder("Привет, ");

sb.Append("Hello");

Console.WriteLine(sb.ToString());

}

}

23. Опишите назначение класса Regex. Приведите перечень категорий символов, операторов и конструкций, которые можно использовать для задания регулярных выражений. Интерпретируйте следующие шаблоны: \b.\*[.?!;:](\s|\z), gr[ae]y\s\S+?[\s|\p{P}], \b[A-Z]\w\*\b, \bth[^o]\w+\b, \w)\1, \b(\w+)(\W){1,2}, \b\w+(e)?s(\s|$), \b(\S+)\s?, ^\D\d{1,5}\D\*$, ^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$, , ^[0-9-[2468]]+$.

Класс Regex в языке C# используется для работы с регулярными выражениями, которые представляют собой шаблоны для поиска и сопоставления текстовых данных.

Перечень категорий символов, операторов и конструкций для задания регулярных выражений включает:

* Категории символов: \d (цифры), \w (буквы и цифры), \s (пробельные символы), \p{P} (знаки пунктуации) и другие.
* Операторы: . (любой символ), \* (ноль или более повторений), + (одно или более повторений), ? (ноль или одно повторение), [...] (символьный класс) и другие.
* Конструкции: \b (граница слова), ^ (начало строки), $ (конец строки), () (группировка), | (или) и другие.

Интерпретация шаблонов:

1. \b.\*[.?!;:](\s|\z) - находит последнее предложение в тексте, заканчивающееся на символы ., ?, !, ; или :, с последующим пробелом или концом строки.
2. gr[ae]y\s\S+?[\s|\p{P}] - находит слова "gray" или "grey", за которыми следует пробел, а затем один или более не-пробельных символов, завершающихся пробелом или знаком пунктуации.
3. \b[A-Z]\w\*\b - находит слова, начинающиеся с заглавной буквы.
4. \bth[^o]\w+\b - находит слова, начинающиеся с букв "th" и не заканчивающиеся на "o".
5. \w)\1 - находит повторяющиеся пары символов, состоящие из буквы или цифры, за которой следует та же самая буква или цифра.
6. \b(\w+)(\W){1,2} - находит слово или набор слов, за которым следует один или два не-буквенных символа.
7. \b\w+(e)?s(\s|$) - находит слова во множественном числе, оканчивающиеся на "s" или "es" и за которыми следует пробел или конец строки.
8. \b(\S+)\s? - находит последнее слово в строке.
9. ^\D\d{1,5}\D\*$ - находит строки, начинающиеся с нецифрового символа, за которым следует от 1 до 5 цифр, а затем могут быть нецифровые символы до конца строки.
10. ^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$ - находит строки, представляющие собой телефонные номера в формате (xxx) xxx-xxxx или xxx-xxx-xxxx, где x - цифры.
11. ^[0-9-[2468]]+$ - находит строки, состоящие только из цифр, исключая четные числа.

Обратите внимание, что описанные интерпретации шаблонов могут быть приближенными и могут иметь особенности в зависимости от контекста использования регулярного выражения.

24. Опишите методы для работы с регулярными выражениями в С#. Перечислите основные управляющие символы для построения регулярных выражений. Постройте регулярное выражение для вывода IP-адреса и телефонного номера в формате +375(29|44|25|17)ххх-хх-хх.

Методы для работы с регулярными выражениями в C#:

* Regex.IsMatch(input, pattern) - метод, который проверяет, соответствует ли входная строка заданному шаблону регулярного выражения.
* Regex.Match(input, pattern) - метод, который ищет первое вхождение подстроки во входной строке, соответствующее заданному шаблону регулярного выражения.
* Regex.Matches(input, pattern) - метод, который находит все вхождения подстрок, соответствующих заданному шаблону регулярного выражения, во входной строке.
* Regex.Replace(input, pattern, replacement) - метод, который заменяет все вхождения подстрок, соответствующих заданному шаблону регулярного выражения, во входной строке указанной заменой.

Основные управляющие символы для построения регулярных выражений:

* . - соответствует любому символу, кроме символа новой строки.
* \* - соответствует нулю или более повторений предыдущего символа или выражения.
* + - соответствует одному или более повторений предыдущего символа или выражения.
* ? - соответствует нулю или одному повторению предыдущего символа или выражения.
* [] - задает класс символов, соответствующих любому символу внутри скобок.
* () - задает группу символов, которые могут быть повторены или в которых может быть применен оператор.

Регулярное выражение для вывода IP-адреса и телефонного номера в указанном формате:

using System;

using System.Text.RegularExpressions;

class Program

{

static void Main(string[] args)

{

string input = "IP: 192.168.0.1, Phone: +375291234567";

string ipPattern = @"\b(?:\d{1,3}\.){3}\d{1,3}\b";

string phonePattern = @"\+375(?:29|44|25|17)\d{3}-\d{2}-\d{2}";

MatchCollection ipMatches = Regex.Matches(input, ipPattern);

MatchCollection phoneMatches = Regex.Matches(input, phonePattern);

foreach (Match match in ipMatches)

{

Console.WriteLine("IP-адрес: " + match.Value);

}

foreach (Match match in phoneMatches)

{

Console.WriteLine("Телефонный номер: " + match.Value);

}

}

}

25. Опишите синтаксис объявления структуры. Перечислите особенности использования и инициализации объектов структур на языке С#. Создайте структуру для описания класса Человек, полями которого будут имя и возраст, укажите конструктор и метод вывода данных о созданном объекте структуры. Продемонстрируйте работу программы в Main.

struct ИмяСтруктуры

{

// Поля структуры

тип имяПоля1;

тип имяПоля2;

// ...

// Конструктор структуры

public ИмяСтруктуры(параметры)

{

// Инициализация полей

// ...

}

// Методы структуры

тип Метод1(параметры)

{

// Тело метода

// ...

}

// ...

}

Особенности использования и инициализации объектов структур в C#:

* Объекты структур являются значимыми типами данных и передаются по значению.
* Поля структур по умолчанию инициализируются значениями по умолчанию для их типов.
* Можно создавать объекты структур с помощью ключевого слова new.
* Можно обращаться к полям структуры с использованием оператора точки (.).
* Можно передавать объекты структур в методы и возвращать их из методов.

Пример создания структуры для описания класса "Человек":

struct Person

{

public string Name;

public int Age;

public Person(string name, int age)

{

Name = name;

Age = age;

}

public void Display()

{

Console.WriteLine("Name: " + Name);

Console.WriteLine("Age: " + Age);

}

}

Пример использования созданной структуры в методе Main:

static void Main()

{

Person person = new Person("John", 25);

person.Display();

}

26. Для чего предназначена структура DateTime? Перечислите виды форматов даты. Перечислите ее основные методы и свойства. Считайте с компьютера текущую дату и найдите полное количество лет человеку, у которого день рождения 23.08.2000.

Структура DateTime в C# предназначена для работы с датами и временем.

Виды форматов даты:

* "d": короткий формат даты (например, 6/15/2009).
* "D": длинный формат даты (например, Monday, June 15, 2009).
* "t": короткий формат времени (например, 1:45 PM).
* "T": длинный формат времени (например, 1:45:30 PM).
* "f": полный формат даты и времени (например, Monday, June 15, 2009 1:45 PM).
* "F": полный формат даты и времени с полной секундой (например, Monday, June 15, 2009 1:45:30 PM).
* "g": общий формат даты и времени (например, 6/15/2009 1:45 PM).
* "G": общий формат даты и времени с полной секундой (например, 6/15/2009 1:45:30 PM).
* "M": формат даты без времени (например, June 15).
* "Y": формат месяца и года (например, June, 2009).

Некоторые основные методы и свойства структуры DateTime:

* Now: свойство, возвращающее текущую дату и время.
* Today: свойство, возвращающее текущую дату без времени.
* Parse: метод, преобразующий строку в объект DateTime.
* ToString: метод, преобразующий объект DateTime в строку с определенным форматом.
* Add: метод, добавляющий определенное значение времени или даты к объекту DateTime.
* Subtract: метод, вычисляющий разницу между двумя объектами DateTime.
* Year, Month, Day, Hour, Minute, Second: свойства, позволяющие получить отдельные компоненты даты и времени.

Пример для подсчета полного количества лет для человека с днем рождения 23.08.2000:

DateTime birthday = new DateTime(2000, 8, 23);

DateTime today = DateTime.Today;

int age = today.Year - birthday.Year;

if (today < birthday.AddYears(age))

{

age--;

}

Console.WriteLine("Возраст: " + age + " лет");

27. Дайте определение понятиям «класс» и «объект». Перечислите элементы, которые могут содержаться в классе. Для чего используется ключевое слово this? Приведите пример использования this в конструкторе и в методе класса

"Класс" - это шаблон, описание состоящий из методов, свойств и данных, определяющий поведение объектов. "Объект" - это экземпляр класса, созданный на основе его шаблона, обладающий своим уникальным состоянием и поведением.

**Элементы, которые могут содержаться в классе**

Класс может содержать следующие элементы:

* Поля (fields), отображающие состояние объекта класса (например, private int x).
* Свойства (properties), которые обеспечивают доступ к полям класса (например, public int X { get { return x; } set { x = value; } })
* Методы (methods), определяющие поведение объекта (например, public void Draw())
* Конструкторы (constructors), устанавливающие начальные значения полей класса.
* Деструкторы (destructors), выполняющие очистку памяти, используемой объектом класса.
* События (events), определяющие оповещения об изменениях в классе.
* Вложенные классы (nested classes).

**Использование ключевого слова this**

"this" используется для ссылки на текущий экземпляр объекта класса. Он используется, когда нужно явно указать, на какой именно объект класса выполняются определенные действия или операции, особенно в тех случаях, когда имена членов класса совпадают с именами параметров метода или конструктора.

**Пример использования this в конструкторе и в методе класса**

public class Rectangle{

private int width;

private int height;

// Конструктор класса, использующий this для установки значений полей

public Rectangle(int width, int height) {

this.width = width;

this.height = height;

}

// Метод, который использует поля класса и возвращает результат

public int GetArea() {

return this.width \* this.height;

}

}

В приведенном примере конструктор класса использует ключевое слово this, чтобы установить начальные значения полям width и height. Метод GetArea также использует ключевое слово this, чтобы получить доступ к полям класса для вычисления площади прямоугольника.

28. Из чего состоит UML-представление класса? Перечислите и опишите виды связей между классами. Постройте диаграмму классов с полями, методами, событиями, свойствами и др. для классов Колледж, Диплом, Студент, Преподаватель, Оценка, Экзамен, Пересдача, Зав.Отделения, Отчисление, Армия.

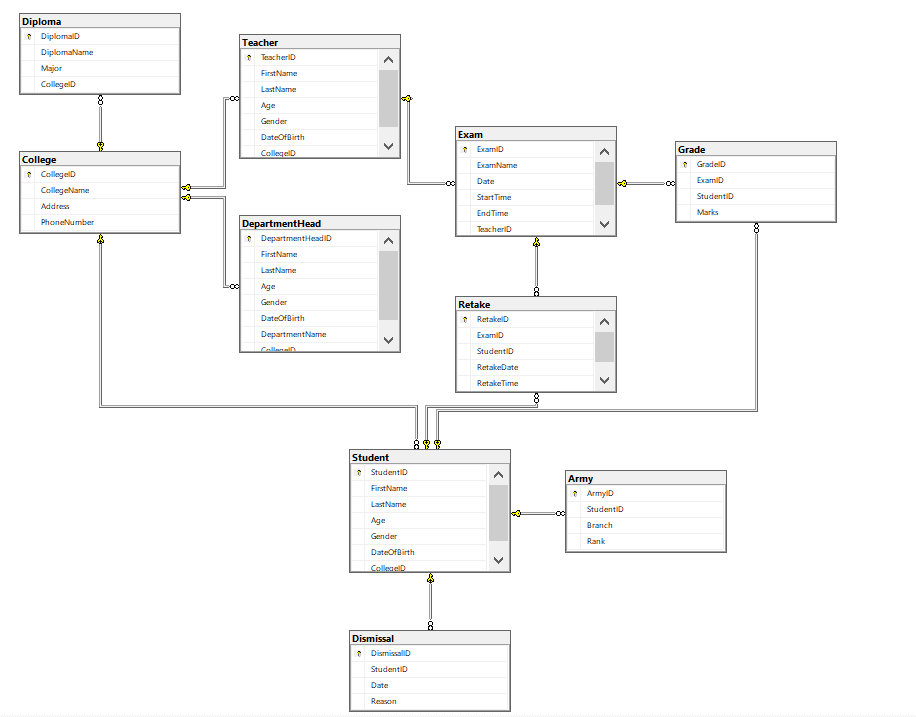
Класс на UML-диаграмме состоит из:

* Названия класса (обычно в верхней части квадрата)
* Атрибутов класса (поля, свойства)
* Операций класса (методы)
* Видимости атрибутов и операций (public, private и т.д.)
* Списка базовых классов
* Имеет ли класс абстрактный базовый класс или интерфейс.
* Списка зависимых классов

**Виды связей между классами**

На UML-диаграммах классов принято выделять несколько видов связей между классами:

1. Наследование (Inheritance) - это связь между классами, когда один класс наследует часть свойств и/или методов у другого класса (базовый класс). Наследование обозначается стрелкой со звездочкой на конце, указывающей на базовый класс.
2. Ассоциация (Association) - это связь между двумя классами, когда один класс использует другой класс. Ассоциация обозначается линией со стрелкой, указывающей на использованный класс.
3. Агрегация (Aggregation) - это связь между целым и его частями, когда объект целого включает в себя один или несколько объектов-частей. Агрегация обозначается линией со стрелкой, указывающей на часть. Другим вариантом обозначения агрегации является линия со стрелкой, указывающей на часть, и графически привязанный к целому диск.
4. Композиция (Composition) - это связь между целым и его частями, когда объект части существует только в рамках объекта целого, и удаляется одновременно с ним. Композиция обозначается линией со стрелкой, указывающей на часть, и графически привязанный к целому заливкой.



29. Основными принципами ООП являются:

1. Инкапсуляция (Encapsulation) - это механизм, позволяющий объединить данные и методы в одном классе и скрыть внутреннее состояние объекта от прямого доступа извне. Цель инкапсуляции - обеспечить защиту данных от изменения извне и обеспечить доступ к этим данным только через методы класса.
2. Наследование (Inheritance) - это механизм, позволяющий создавать новый класс на основе существующего, наследуя его свойства и методы. Цель наследования - повторно использовать код и обеспечить многократное использование уже существующих решений.
3. Полиморфизм (Polymorphism) - это механизм, который позволяет использовать объекты разных классов с единым интерфейсом. Цель полиморфизма - достижение универсальности программирования, т.е. позволяет создавать более гибкие и расширяемые программы.

**Иерархия классов**

abstract class Figure

{

public abstract void PrintInfo();

}

class Square : Figure

{

private int a;

public Square(int a) {

this.a = a;

}

public override void PrintInfo()

{

Console.WriteLine("Square with side length {0}", a);

}

}

class Rectangle : Square

{

private int b;

public Rectangle(int a, int b) : base(a)

{

this.b = b;

}

public override void PrintInfo()

{

Console.WriteLine("Rectangle with side lengths {0} and {1}", a, b);

}

}

**Демонстрация работы классов**

class Program

{

static void Main(string[] args)

{

Figure f = new Square(5);

f.PrintInfo(); // Output: Square with side length 5

Figure f2 = new Rectangle(3, 4);

f2.PrintInfo(); // Output: Rectangle with side lengths 3 and 4

}

}

В этом примере создается иерархия классов: абстрактный класс Figure, его наследник Square и его наследник Rectangle

30. Перечислите принципы SOLID. Опишите каждый из принципов и укажите его назначение. Приведите примеры на каждый из принципов, укажите их преимущества и недостатки.

Принципы SOLID являются руководством для разработки стабильных, гибких и поддерживаемых программных систем. SOLID состоит из следующих принципов:

1. Принцип единственной ответственности (Single Responsibility Principle, SRP

Этот принцип утверждает, что каждый класс или модуль должен иметь только одну причину для изменения. Он должен быть ответственным только за одну аспект функциональности программы. Это помогает обеспечить легкую поддержку, повторное использование кода и улучшает понимание классов.

Пример: Разделение класса, отвечающего за хранение данных (DataStorage) и класса, отвечающего за работу с этими данными (DataProcessor). Преимущества включают легкую поддержку и модификацию каждого класса независимо от другого. Недостаток - возможное увеличение числа классов.

2. Принцип открытости/закрытости (Open/Closed Principle, OCP

Этот принцип указывает, что классы должны быть открыты для расширения, но закрыты для модификации. Это означает, что изменения в функциональности должны добавляться путем создания нового кода, не внося изменения в существующий. Такой подход позволяет избежать непредвиденных сайд-эффектов.

Пример: Создание базового класса (Shape) и наследование от него для создания новых фигур (Circle, Square). Преимущество заключается в возможности добавления новых фигур, не изменяя существующие. Недостаток - необходимость создания дополнительных классов и их наследование.

3. Принцип подстановки Барбары Лисков (Liskov Substitution Principle, LSP

Согласно этому принципу, подклассы должны быть взаимозаменяемыми с их базовым классом. То есть, код, который использует базовый класс, должен корректно работать и с подклассами, не зная о них. Это помогает обеспечить правильность и надежность программы.

Пример: Создание класса отображения фигур (ShapeRenderer) и его использование для отображения как круга, так и квадрата. Преимущества включают возможность расширения класса без влияния на код, который использует его. Недостаток - потенциальное нарушение принципа, если подклассы не могут заменять базовый класс.

4. Принцип разделения интерфейса (Interface Segregation Principle, ISP

Этот принцип говорит о том, что клиенты не должны зависеть от интерфейсов, которые они не используют. Лучше создать несколько специализированных интерфейсов, чем один общий, чтобы избежать накопления излишних зависимостей и уменьшить связанность между классами.

Пример: Разделение общего интерфейса для работы с животными (Animal) на отдельные интерфейсы для летающих (Flying), плавающих (Swimming) и бегающих (Running) животных. Преимущество - уменьшение связанности и гибкость при внесении изменений. Недостаток - увеличение числа интерфейсов.

5. Принцип инверсии зависимости (Dependency Inversion Principle, DIP

Этот принцип гласит, что модули верхнего уровня не должны зависеть от модулей нижнего уровня. Они должны зависеть от абстракций, а не от конкретной реализации. Это позволяет легко вносить изменения и заменять компоненты.

Пример: Использование внедрения зависимостей (Dependency Injection) для связи классов. Преимущества включают возможность легко заменить реализацию класса без необходимости изменения высокоуровневых модулей. Недостаток - возможное усложнение конфигурации и введение дополнительного слоя абстракции.

Каждый из принципов SOLID дает преимущества в разработке гибких и поддерживаемых систем, но их применение требует внимательного анализа конкретной ситуации и учета контекста разработки.

31. Опишите назначение инкапсуляции. Перечислите все возможные спецификаторы доступа к пользовательским классам С# с пояснением их назначения. Укажите спецификатор доступа для полей класса, который используется по умолчанию (если он не указан явно). На примере класса Треугольник, для которого можно рассчитать площадь, покажите использование различных типов доступа.

Назначение инкапсуляции в объектно-ориентированном программировании заключается в объединении данных и методов, работающих с этими данными, в единый компонент (класс), и скрытии внутренних деталей реализации от внешнего кода. Это обеспечивает контролируемый доступ к данным и повышает безопасность, надежность и поддерживаемость кода.

В языке С# используются следующие спецификаторы доступа к классам и их членам:

1. public Члены с модификатором `public` доступны из любого места в программе. Они могут быть использованы как внутри, так и вне класса.

2. private Члены с модификатором `private` доступны только внутри того же класса. Они скрыты от других классов и обеспечивают приватность данных и методов класса.

3. protected Члены с модификатором `protected` доступны внутри того же класса и его производных классов (наследников). Они не доступны извне класса и служат для обеспечения контролируемого доступа в иерархии наследования.

4. internal Члены с модификатором `internal` доступны внутри текущей сборки (assembly). Они не могут быть использованы из других сборок.

5. protected internal Члены с модификатором `protected internal` доступны внутри текущей сборки и внутри производных классов (наследников) даже в других сборках.

При отсутствии явного указания спецификатора доступа у полей класса, используется спецификатор доступа `private` по умолчанию.

public class Triangle

{

private double sideA; // Приватное поле

public double SideB { get; set; } // Публичное свойство со свободным доступом к чтению и записи

protected double SideC { get; set; } // Защищенное свойство

internal double CalculateArea() // Внутренний метод для расчета площади

{

// Реализация расчета площади

}

}

Здесь поле `sideA` объявлено с модификатором доступа `private`, что означает, что оно может быть использовано только внутри класса. Свойство `SideB` имеет модификатор доступа `public`, поэтому оно доступно как внутри класса, так и во внешнем коде. Свойство `SideC` с модификатором доступа `protected` может быть использовано внутри текущего класса и его производных классов. Метод `CalculateArea` имеет модификатор доступа `internal`, поэтому он может использоваться только внутри текущей сборки.

32. Дайте определение понятию «конструктор». Перечислите особенности конструктора класса. Опишите процесс использования конструктора при создании объекта класса Напишите фрагмент программы описания класса “Прямоугольник”, методами которого представить перегрузку конструкторов. Продемонстрируйте в Main работу с перегруженными конструкторами.

Конструктор - это специальный метод класса, который вызывается при создании нового объекта данного класса. Он используется для инициализации членов объекта и выполнения других необходимых операций перед его использованием.

Особенности конструктора класса:

1. Имя конструктора должно совпадать с именем класса.

2. Он не возвращает значения, даже void.

3. Конструктор может быть перегружен, то есть класс может иметь несколько конструкторов с разными параметрами.

4. Конструктор вызывается автоматически при создании объекта с помощью оператора `new`.

5. Конструктор может содержать инициализацию переменных класса и вызов других методов класса.

Процесс использования конструктора при создании объекта класса включает следующие шаги:

1. Объявление переменной класса с указанием типа и имени.

2. Использование оператора `new` с вызовом конструктора класса и передачей аргументов, если необходимо.

3. Присвоение созданного объекта переменной класса.

4. Выполнение операций с объектом и его членами.

class Rectangle

{

private int width;

private int height;

// Первый конструктор без параметров

public Rectangle()

{

width = 0;

height = 0;

}

// Второй конструктор с параметрами

public Rectangle(int w, int h)

{

width = w;

height = h;

}

// Метод для получения площади прямоугольника

public int GetArea()

{

return width \* height;

}

}

Продемонстрируем работу с перегруженными конструкторами в методе `Main`:

static void Main(string[] args)

{

// Создание объекта с помощью первого конструктора

Rectangle rect1 = new Rectangle();

Console.WriteLine("Ширина прямоугольника: " + rect1.Width); // Вывод: 0

Console.WriteLine("Высота прямоугольника: " + rect1.Height); // Вывод: 0

// Создание объекта с помощью второго конструктора

Rectangle rect2 = new Rectangle(5, 10);

Console.WriteLine("Ширина прямоугольника: " + rect2.Width); // Вывод: 5

Console.WriteLine("Высота прямоугольника: " + rect2.Height); // Вывод: 10

// Вызов метода для получения площади прямоугольника

int area = rect2.GetArea();

Console.WriteLine("Площадь прямоугольника: " + area); // Вывод: 50

}

33. Опишите синтаксис свойств и укажите назначение специальных методов (асессоров) get и set. Опишите процесс использования свойств. Для класса Человек с полями имя и возраст осуществите с помощью set и get проверку вводимых данных (имя – с большой буквы, возраст – от 0 до 100) и измените возраст +10.

Свойства (Properties) в языке программирования C# представляют собой специальные методы (асессоры) - get и set, которые позволяют управлять доступом к полям класса и обеспечивают возможность проверки и изменения значений переданных данных.

Синтаксис свойства в C# выглядит следующим образом:

```csharp

тип\_данных имя\_свойства

{

get { /\* код для получения значения свойства \*/ }

set { /\* код для установки значения свойства \*/ }

}

```

- `get` - метод-аксессор для получения значения свойства. Он возвращает значение свойства и не имеет параметров. Используйте ключевое слово `return` для возврата значения.

- `set` - метод-аксессор для установки значения свойства. Он принимает параметр, обозначаемый ключевым словом `value`, и не возвращает значения. Внутри этого метода можно добавить логику проверки и изменения значения перед его установкой в поле.

Процесс использования свойств включает следующие шаги:

1. Объявление свойства внутри класса.

2. Задание логики для `get` и `set` методов, включая проверку и изменение данных при необходимости.

3. Использование свойства при доступе к данным класса, как если бы это были обычные поля.

Пример использования свойств для класса "Человек" с полями "имя" и "возраст" для проверки вводимых данных и изменения возраста на +10:

```csharp

class Person

{

private string name;

private int age;

public string Name

{

get { return name; }

set { name = char.ToUpper(value[0]) + value.Substring(1); }

}

public int Age

{

get { return age; }

set

{

if (value >= 0 && value <= 100)

age = value;

else

throw new ArgumentOutOfRangeException("Age must be between 0 and 100.");

}

}

}

```

В приведенном примере свойство `Name` проверяет, что первая буква имени будет большой, а все остальные - маленькими буквами. Свойство `Age` проверяет, что возраст находится в диапазоне от 0 до 100, и, при несоответствии данному диапазону, генерирует исключение `ArgumentOutOfRangeException`.

Продемонстрируем процесс использования свойств и изменение возраста +10 в методе `Main`:

```csharp

static void Main(string[] args)

{

Person person = new Person();

person.Name = "john doe";

person.Age = 25;

Console.WriteLine("Name: " + person.Name); // Выводит: John doe

Console.WriteLine("Age: " + person.Age); // Выводит: 25

person.Age += 10;

Console.WriteLine("New Age: " + person.Age); // Выводит: 35

}

```

Таким образом, пример демонстрирует использование свойств `Name` и `Age`, проверку данных и изменение возраста на +10.

34. Наследование в объектно-ориентированном программировании позволяет создавать иерархию классов, где классы наследуют свойства и методы от других классов. Класс, от которого происходит наследование, называется базовым классом или суперклассом, а класс, который наследует эти свойства и методы, называется производным классом или подклассом.

Абстрактный класс - это класс, который не может быть создан в виде объекта и предназначен для использования в качестве базового класса. Абстрактные классы содержат как реализацию методов, так и объявления абстрактных методов. Абстрактный метод не имеет определения в базовом классе, но должен быть реализован в производном классе.

Особенности абстрактных классов:

1. Нельзя создать экземпляр абстрактного класса, он может использоваться только в качестве базового класса для других классов.

2. Абстрактные классы могут содержать реализацию методов, а также объявления абстрактных методов, которые должны быть реализованы в производных классах.

3. Класс, наследующий от абстрактного класса, должен реализовать все абстрактные методы из базового класса.

Пример описания абстрактного класса и производного от него класса на языке C#:

```csharp

abstract class Shape

{

public abstract void Draw();

}

class Circle : Shape

{

public override void Draw()

{

Console.WriteLine("Drawing a circle");

}

}

```

В данном примере у нас есть абстрактный класс `Shape`, который содержит один абстрактный метод `Draw()`. Метод `Draw()` не имеет реализации в абстрактном классе, поэтому класс `Shape` также является абстрактным. Затем у нас есть производный класс `Circle`, который наследует от `Shape` и реализует абстрактный метод `Draw()`, чтобы нарисовать круг.

Пример создания конкретного объекта:

```csharp

Shape shape = new Circle();

shape.Draw(); // Выводит "Drawing a circle"

```

В этом примере мы создаем объект `Circle`, который является производным классом от `Shape`, и сохраняем его в переменной типа `Shape`. Затем мы вызываем метод `Draw()` для этого объекта, который выводит "Drawing a circle" на консоль.

35. Опишите назначение полиморфизма. Укажите назначение ключевых слов virtual и override. Опишите особенности использования виртуальных методов. Продемонстрируйте работу виртуальных методов для расчета площади на примере иерархии фигур Квадрат -> Прямоугольник с виртуальным методом площадь.

Полиморфизм в объектно-ориентированном программировании позволяет объектам разных типов обращаться к одному и тому же методу, но давать различные результаты в зависимости от типа объекта, с которым метод вызывается. Это означает, что метод может иметь разные реализации для разных классов, но вызываться с использованием общего интерфейса.

Ключевые слова `virtual` и `override` в C# используются для реализации полиморфизма через виртуальные методы.

- `virtual` - ключевое слово, которое указывает, что метод в базовом классе может быть переопределен (изменен) в производном классе. Оно позволяет создать виртуальный метод с реализацией по умолчанию в базовом классе.

- `override` - ключевое слово, которое указывает, что метод в производном классе переопределяет (изменяет) реализацию виртуального метода, определенного в базовом классе.

Особенности использования виртуальных методов:

1. Виртуальные методы должны быть определены в базовом классе с использованием ключевого слова `virtual`.

2. Производный класс должен использовать ключевое слово `override`, чтобы указать, что метод переопределяет виртуальный метод базового класса.

3. При вызове виртуального метода для объекта производного класса будет вызываться переопределенная версия метода из производного класса, даже если переменная ссылается на базовый класс.

Пример использования виртуальных методов для расчета площади на примере иерархии фигур "Квадрат" -> "Прямоугольник":

```csharp

using System;

class Shape

{

public virtual double CalculateArea()

{

return 0; // Базовая реализация возвращает 0

}

}

class Square : Shape

{

public double SideLength { get; set; }

public override double CalculateArea()

{

return SideLength \* SideLength; // Переопределение для квадрата

}

}

class Rectangle : Shape

{

public double Width { get; set; }

public double Height { get; set; }

public override double CalculateArea()

{

return Width \* Height; // Переопределение для прямоугольника

}

}

class Program

{

static void Main(string[] args)

{

Shape shape1 = new Square() { SideLength = 5 };

Shape shape2 = new Rectangle() { Width = 4, Height = 6 };

Console.WriteLine("Площадь квадрата: " + shape1.CalculateArea()); // Выводит "Площадь квадрата: 25"

Console.WriteLine("Площадь прямоугольника: " + shape2.Calculate

Area()); // Выводит "Площадь прямоугольника: 24"

}

}

```

В этом примере у нас есть базовый класс `Shape`, в котором определен виртуальный метод `CalculateArea()`, возвращающий 0. Затем у нас есть два производных класса: `Square` (квадрат) и `Rectangle` (прямоугольник), которые переопределяют метод `CalculateArea()` для расчета площади соответствующей фигуры.

В методе `Main` мы создаем объекты `Square` и `Rectangle` и присваиваем их переменным типа `Shape`. При вызове метода `CalculateArea()` для каждого объекта будет вызываться соответствующая переопределенная версия метода, и будет выведен результат расчета площади на консоль.

1. Укажите назначение операторов для обработки исключений try-catch-finally. Опишите процесс создания пользовательского класса исключений. Укажите назначение оператора throw. Опишите класс Оценка за семестр с полем оценка (int), конструктором с параметром, свойством, проверяющим, что оценка лежит в диапазоне от 0 до 10, и методом вывода информации об объекте. Реализуйте свой класс для обработки исключений для данного класса. Продемонстрируйте работоспособность разработанной программы в Main.

Операторы try-catch-finally используются для обработки исключений в C#. Их назначение состоит в следующем:

try: В блоке try размещается код, который потенциально может вызвать исключение. Если исключение происходит в блоке try, выполнение кода в этом блоке прерывается, и управление передается соответствующему блоку catch.

catch: В блоке catch указывается код для обработки исключения. Блок catch выполняется только в случае, если произошло исключение в блоке try. Он содержит тип исключения, который может быть обработан, и соответствующий код для обработки исключения.

finally: Блок finally содержит код, который выполняется всегда, независимо от того, происходило исключение или нет. Он используется для выполнения завершающих действий, таких как освобождение ресурсов, независимо от того, возникло исключение или нет.

Процесс создания пользовательского класса исключений включает в себя следующие шаги:

Создание класса, который наследуется от базового класса Exception или его производных классов.

Определение конструкторов класса для инициализации объекта исключения.

Предоставление дополнительных свойств и методов для получения дополнительной информации об исключении.

Например, рассмотрим класс Grade (Оценка) с полем score (оценка), конструктором с параметром, свойством Score, проверяющим, что оценка лежит в диапазоне от 0 до 10, и методом PrintInfo, выводящим информацию об объекте:

csharp

Copy code

public class Grade

{

private int score;

public int Score

{

get { return score; }

set

{

if (value >= 0 && value <= 10)

score = value;

else

throw new GradeOutOfRangeException("Оценка должна быть в диапазоне от 0 до 10.");

}

}

public Grade(int score)

{

Score = score;

}

public void PrintInfo()

{

Console.WriteLine($"Оценка: {score}");

}

}

Для обработки исключений в данном классе можно создать пользовательский класс исключений GradeOutOfRangeException, наследующийся от Exception, чтобы обрабатывать ситуации, когда оценка выходит за диапазон от 0 до 10:

csharp

Copy code

public class GradeOutOfRangeException : Exception

{

public GradeOutOfRangeException(string message) : base(message)

{

}

}

Пример использования класса Grade с обработкой исключений:

csharp

Copy code

try

{

Grade grade = new Grade(8);

grade.PrintInfo();

grade.Score = 15; // Генерируется исключение GradeOutOfRangeException

}

catch (GradeOutOfRangeException ex)

{

Console.WriteLine($"Исключение: {ex.Message}");

}

1. Дайте определение понятию «делегат». Укажите синтаксис объявления делегата, создание экземпляров делегата и их использование. Перечислите операции, которые можно использовать с делегатами. Приведите пример передачи делегата в качестве параметра в функцию.

Делегат в C# представляет тип, который инкапсулирует ссылку на метод. Он позволяет передавать методы как параметры, возвращать методы из других методов и сохранять ссылки на методы в коллекциях.

Синтаксис объявления делегата:

delegate void MyDelegate(int x); // объявление делегата

Создание экземпляров делегата и их использование:

public class MyClass

{

public void MyMethod(int x)

{

Console.WriteLine($"Значение x: {x}");

}

}

public class Program

{

public static void Main()

{

MyClass obj = new MyClass();

MyDelegate del = new MyDelegate(obj.MyMethod);

del(5); // Вызов делегата, что приведет к вызову метода MyMethod с параметром 5

}

}

Операции, которые можно использовать с делегатами, включают:

Присваивание методов делегату.

Комбинирование делегатов с использованием оператора +.

Удаление методов из делегата с использованием оператора -.

Вызов делегата, вызывающего все методы, связанные с ним.

Пример передачи делегата в качестве параметра в функцию:

public delegate void MyDelegate(int x);

public class MyClass

{

public void MyMethod(int x)

{

Console.WriteLine($"Значение x: {x}");

}

public void ProcessDelegate(MyDelegate del)

{

del(10); // Вызов делегата

}

}

public class Program

{

public static void Main()

{

MyClass obj = new MyClass();

MyDelegate del = new MyDelegate(obj.MyMethod);

obj.ProcessDelegate(del); // Передача делегата в качестве параметра

}

}

1. Дайте определение понятию «событие» и «обработчик события». Приведите синтаксис события. Укажите как события связано с делегатом. Создайте класс Мешок, полем которого будет количество шаров. Создайте событие добавления и удаления шаров из мешка, предусматривающих различные случаи. Продемонстрируйте работу с событиями в Main.

В C# событие представляет действие или происшествие, которое может возникнуть внутри класса, и обработчик события представляет метод, который выполняется при наступлении события. События позволяют связать различные части кода, чтобы реагировать на определенные ситуации или взаимодействия.

Синтаксис события:

public event EventHandler MyEvent;

Событие MyEvent является экземпляром класса делегата EventHandler, который представляет методы, принимающие два параметра: объект, генерирующий событие, и аргументы события.

Пример класса Bag (Мешок) событий добавления и удаления шаров:

public class BallEventArgs : EventArgs

{

public int BallCount { get; }

public BallEventArgs(int ballCount)

{

BallCount = ballCount;

}

}

public class Bag

{

private int ballCount;

public event EventHandler<BallEventArgs> BallAdded;

public event EventHandler<BallEventArgs> BallRemoved;

public int BallCount

{

get { return ballCount; }

set

{

if (value >= 0)

{

int diff = value - ballCount;

if (diff > 0)

OnBallAdded(diff);

else if (diff < 0)

OnBallRemoved(-diff);

ballCount = value;

}

}

}

protected virtual void OnBallAdded(int count)

{

BallAdded?.Invoke(this, new BallEventArgs(count));

}

protected virtual void OnBallRemoved(int count)

{

BallRemoved?.Invoke(this, new BallEventArgs(count));

}

}

public class Program

{

public static void Main()

{

Bag bag = new Bag();

bag.BallAdded += Bag\_BallAdded;

bag.BallRemoved += Bag\_BallRemoved;

bag.BallCount = 5; // Выведет: Шары добавлены: 5

bag.BallCount = 2; // Выведет: Шары удалены: 3

}

private static void Bag\_BallAdded(object sender, BallEventArgs e)

{

Console.WriteLine($"Шары добавлены: {e.BallCount}");

}

private static void Bag\_BallRemoved(object sender, BallEventArgs e)

{

Console.WriteLine($"Шары удалены: {e.BallCount}");

}

}

1. Дайте определение понятию «анонимный метод». Укажите чем лямбда-выражение отличается от анонимного метода. Приведите синтаксис анонимного метода и лямбда-выражения. Напишите сортировку массива по убыванию с использованием лямбда-выражения.

Анонимный метод в C# представляет безымянный блок кода, который может быть присвоен делегату или использован как аргумент метода. Лямбда-выражение является упрощенной формой анонимного метода, которое предоставляет компактный синтаксис для создания делегатов.

Синтаксис анонимного метода:

delegate void MyDelegate(int x);

public class Program

{

public static void Main()

{

MyDelegate del = delegate (int x) // анонимный метод

{

Console.WriteLine($"Значение x: {x}");

};

del(5); // Вызов делегата с анонимным методом

}

}

Синтаксис лямбда-выражения:

delegate void MyDelegate(int x);

public class Program

{

public static void Main()

{

MyDelegate del = (x) => // лямбда-выражение

{

Console.WriteLine($"Значение x: {x}");

};

del(5); // Вызов делегата с лямбда-выражением

}

}

Пример сортировки массива по убыванию с использованием лямбда-выражения:

int[] numbers = { 5, 2, 8, 1, 9 };

Array.Sort(numbers, (x, y) => y.CompareTo(x));

foreach (int number in numbers)

{

Console.WriteLine(number);

}

1. Опишите разницу между List, ArrayList и массивом. Перечислите обобщенные и не обобщенные коллекции. Удалите положительные четные элементы из целочисленного листа.

Разница между List, ArrayList и массивом:

Массив (Array) является статической структурой данных, представляющей непрерывную последовательность элементов одного типа. Размер массива фиксирован и определяется при его создании. Массивы могут быть обобщенными и не обобщенными. Доступ к элементам массива осуществляется по индексу, и они обладают высокой производительностью. Однако изменение размера массива требует создания нового массива.

ArrayList является классом из пространства имен System.Collections, и он представляет динамический массив, в который можно добавлять элементы различных типов. Размер ArrayList автоматически увеличивается при добавлении элементов. Однако ArrayList хранит объекты в виде System.Object, что требует приведения типов при извлечении элементов, что может вызвать ошибки во время выполнения.

List<T> является обобщенным классом, введенным в .NET Framework 2.0, и представляет динамический список элементов определенного типа T. List<T> предоставляет типобезопасность и обладает более высокой производительностью, чем ArrayList. Он имеет аналогичные методы и свойства для работы со списком, такие как Add, Remove, Count и т.д.

Пример удаления положительных четных элементов из целочисленного списка (List<int>):

List<int> numbers = new List<int>() { 1, 2, -3, 4, 5, -6, 7, 8 };

numbers.RemoveAll(x => x > 0 && x % 2 == 0);

foreach (int number in numbers)

{

Console.WriteLine(number);

}

1. Перечислите виды параметризованных коллекций. Опишите коллекцию Dictionary<T,K>. Напишите фрагмент кода с использованием коллекции Dictionary<T,K>.

Виды параметризованных коллекций включают List<T>, Dictionary<TKey, TValue>, Queue<T>, Stack<T>, LinkedList<T>, HashSet<T> и другие. Эти коллекции позволяют хранить и управлять группами элементов одного или разных типов.

Коллекция Dictionary<TKey, TValue> представляет ассоциативный массив, в котором каждый элемент представляет пару ключ-значение. Она использует хэш-таблицу для хранения и быстрого доступа к элементам. TKey представляет тип ключа, а TValue - тип значения.

Пример использования коллекции Dictionary<TKey, TValue>:

Dictionary<string, int> studentScores = new Dictionary<string, int>();

studentScores.Add("Alice", 90);

studentScores.Add("Bob", 80);

studentScores.Add("Charlie", 95);

Console.WriteLine(studentScores["Alice"]); // Выведет: 90

if (studentScores.ContainsKey("Bob"))

{

int score = studentScores["Bob"];

Console.WriteLine($"Bob's score: {score}"); // Выведет: Bob's score: 80

}

foreach (KeyValuePair<string, int> pair in studentScores)

{

Console.WriteLine($"{pair.Key}: {pair.Value}");

}

1. Приведите классификацию файлов в С# с указанием стандартных классов пространства имен System.IO для их обработки. Приведите примеры методов и/или свойств для каждого из классов и укажите их назначение. Приведите примеры записи и чтения текстового файла и байтового файла (на основе потока Stream).

Классификация файлов в C#:

FileStream (пространство имен System.IO) - обеспечивает чтение и запись байтового потока данных в файле.

Примеры методов и свойств: Read, Write, Seek, Position, Length.

StreamReader и StreamWriter (пространство имен System.IO) - обеспечивают чтение и запись текстового потока данных в файле.

Примеры методов и свойств: ReadLine, ReadToEnd, Write, WriteLine, Flush.

File (пространство имен System.IO) - предоставляет статические методы для работы с файлами, такие как создание, удаление, перемещение и т.д.

Примеры методов: Create, Delete, Move, Copy.

Directory (пространство имен System.IO) - предоставляет статические методы для работы с каталогами (папками), такие как создание, удаление, перемещение и т.д.

Примеры методов: CreateDirectory, Delete, Move.

Path (пространство имен System.IO) - предоставляет статические методы для работы с путями файлов и каталогов.

Примеры методов: Combine, GetFileName, GetExtension, GetDirectoryName.

Пример записи и чтения текстового файла:

string filePath = "example.txt";

// Запись в текстовый файл

using (StreamWriter writer = new StreamWriter(filePath))

{

writer.WriteLine("Hello, World!");

}

// Чтение из текстового файла

using (StreamReader reader = new StreamReader(filePath))

{

string line = reader.ReadLine();

Console.WriteLine(line); // Выведет: Hello, World!

}

Пример записи и чтения байтового файла на основе потока Stream:

string filePath = "example.bin";

// Запись в байтовый файл

byte[] dataToWrite = { 1, 2, 3, 4, 5 };

using (FileStream stream = new FileStream(filePath, FileMode.Create))

{

stream.Write(dataToWrite, 0, dataToWrite.Length);

}

// Чтение из байтового файла

byte[] dataRead;

using (FileStream stream = new FileStream(filePath, FileMode.Open))

{

dataRead = new byte[stream.Length];

stream.Read(dataRead, 0, dataRead.Length);

}

foreach (byte data in dataRead)

{

Console.WriteLine(data);

}

1. Перечислите классы для работы с дисками, каталогами, файлами. Укажите их основные функции. Напишите программу, которая выведет все имена и свойства дисков, а также перечень файлов и папок, находящихся на диске С: текущего компьютера.

В C# для работы с дисками, каталогами и файлами можно использовать классы из пространства имен System.IO. Ниже перечислены некоторые из этих классов и их основные функции:

Класс DriveInfo - предоставляет информацию о логических дисках компьютера, такую как имя диска, доступный объем памяти и т.д. Основные функции:

* DriveInfo.GetDrives() - получение списка всех доступных дисков на компьютере.
* DriveInfo.Name - имя диска.
* DriveInfo.TotalSize - общий размер диска.
* DriveInfo.AvailableFreeSpace - доступное свободное пространство на диске.

Класс Directory - предоставляет статические методы для работы с каталогами (папками). Основные функции:

* Directory.CreateDirectory(path) - создание нового каталога по указанному пути.
* Directory.GetDirectories(path) - получение списка подкаталогов в указанном каталоге.
* Directory.GetFiles(path) - получение списка файлов в указанном каталоге.
* Directory.Exists(path) - проверка существования каталога по указанному пути.

Класс File - предоставляет статические методы для работы с файлами. Основные функции:

* File.Create(path) - создание нового файла по указанному пути.
* File.Delete(path) - удаление файла по указанному пути.
* File.Exists(path) - проверка существования файла по указанному пути.
* File.ReadAllText(path) - чтение содержимого файла в виде строки.
* File.WriteAllText(path, contents) - запись указанного текста в файл.

Пример программы, которая выводит все имена и свойства дисков, а также перечень файлов и папок на диске C: текущего компьютера:

using System;

using System.IO;

class Program

{

static void Main()

{

// Получение списка всех дисков

DriveInfo[] drives = DriveInfo.GetDrives();

foreach (DriveInfo drive in drives)

{

Console.WriteLine($"Диск: {drive.Name}");

Console.WriteLine($" Тип диска: {drive.DriveType}");

Console.WriteLine($" Общий объем: {drive.TotalSize} байт");

Console.WriteLine($" Доступное свободное пространство: {drive.AvailableFreeSpace} байт");

// Вывод списка файлов на диске C:

if (drive.Name == "C:\\")

{

Console.WriteLine(" Файлы и папки на диске C:");

string[] files = Directory.GetFiles(drive.Name);

foreach (string file in files)

{

Console.WriteLine($" {file}");

}

string[] directories = Directory.GetDirectories(drive.Name);

foreach (string directory in directories)

{

Console.WriteLine($" {directory}");

}

}

Console.WriteLine();

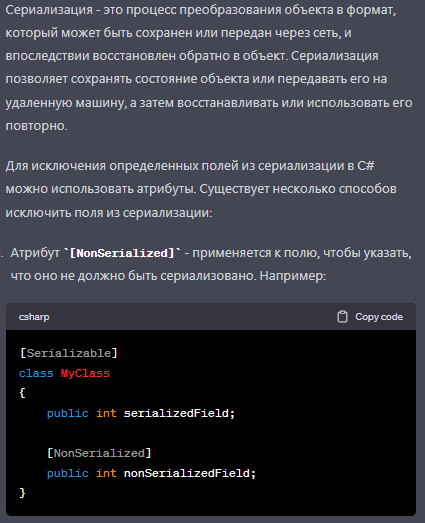
}

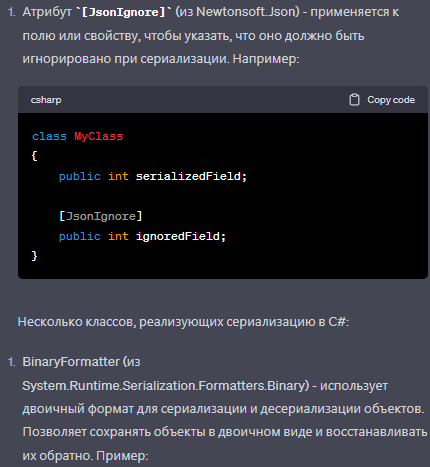
Console.ReadLine();

}

}

1. Что такое сериализация? Как исключить поля из сериализации? Перечислите классы, которые реализуют сериализацию, и укажите их особенности. Приведите примеры.





using System;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

[Serializable]

class MyClass

{

public int value;

}

class Program

{

static void Main()

{

MyClass obj = new MyClass();

obj.value = 42;

BinaryFormatter formatter = new BinaryFormatter();

using (FileStream stream = new FileStream("data.bin", FileMode.Create))

{

formatter.Serialize(stream, obj);

}

using (FileStream stream = new FileStream("data.bin", FileMode.Open))

{

MyClass restoredObj = (MyClass)formatter.Deserialize(stream);

Console.WriteLine(restoredObj.value); // Выводит: 42

}

}

}

XmlSerializer (из System.Xml.Serialization) - позволяет сериализовать объекты в формат XML и десериализовать их обратно. Пример:

using System;

using System.IO;

using System.Xml.Serialization;

[Serializable]

public class MyClass

{

public int value;

}

class Program

{

static void Main()

{

MyClass obj = new MyClass();

obj.value = 42;

XmlSerializer serializer = new XmlSerializer(typeof(MyClass));

using (FileStream stream = new FileStream("data.xml", FileMode.Create))

{

serializer.Serialize(stream, obj);

}

using (FileStream stream = new FileStream("data.xml", FileMode.Open))

{

MyClass restoredObj = (MyClass)serializer.Deserialize(stream);

Console.WriteLine(restoredObj.value); // Выводит: 42

}

}

}

DataContractSerializer (из System.Runtime.Serialization) - используется для сериализации и десериализации объектов в формат XML. Этот сериализатор основан на атрибутах и требует, чтобы классы, подлежащие сериализации, были помечены атрибутами [DataContract] и [DataMember].

Пример:

using System;

using System.IO;

using System.Runtime.Serialization;

[DataContract]

public class MyClass

{

[DataMember]

public int value;

}

class Program

{

static void Main()

{

MyClass obj = new MyClass();

obj.value = 42;

DataContractSerializer serializer = new DataContractSerializer(typeof(MyClass));

using (FileStream stream = new FileStream("data.xml", FileMode.Create))

{

serializer.WriteObject(stream, obj);

}

using (FileStream stream = new FileStream("data.xml", FileMode.Open))

{

MyClass restoredObj = (MyClass)serializer.ReadObject(stream);

Console.WriteLine(restoredObj.value); // Выводит: 42

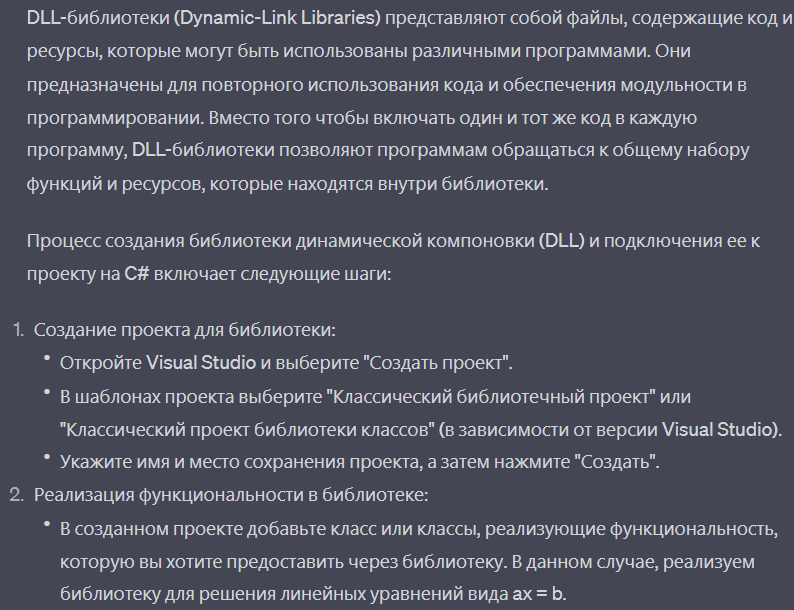
}

}

}

Это только некоторые из классов и методов, предоставляемых в .NET для сериализации объектов. В зависимости от ваших потребностей, вы можете выбрать наиболее подходящий для вас класс и подход к сериализации.

1. Для чего нужны dll-библиотеки? Опишите процесс создания библиотеки динамической компоновки и подключения ее к проекту на С#. Создайте библиотеку для решения линейных уравнений вида ах=b и продемонстрируйте ее подключение и работу в Main.



namespace LinearEquationLibrary

{

public static class LinearEquationSolver

{

public static double Solve(double a, double b)

{

if (a != 0)

{

return b / a;

}

else

{

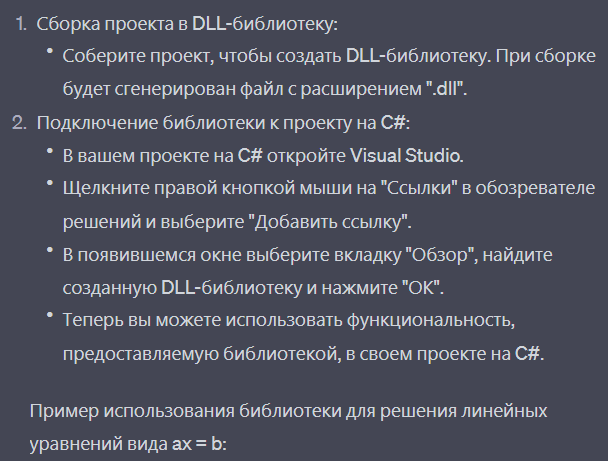
throw new ArgumentException("The coefficient 'a' must not be zero.");

}

}

}

}



using System;

using LinearEquationLibrary; // Подключение библиотеки

class Program

{

static void Main()

{

double a = 2;

double b = 6;

double x = LinearEquationSolver.Solve(a, b); // Использование функции из библиотеки

Console.WriteLine($"Решение уравнения {a}x = {b} равно x = {x}");

Console.ReadLine();

}

}

1. Опишите модель документа XML. Перечислите основные методы и свойства для обращения к элементам XML-документа. Приведите пример создания XML-документа через класс XmlDocument.

Модель документа XML (XML Document Object Model, XML DOM) представляет собой программное представление XML-документа в виде иерархической структуры, состоящей из узлов. XML DOM обеспечивает набор методов и свойств для доступа, создания, модификации и удаления элементов XML-документа.

Основные методы и свойства для обращения к элементам XML-документа с использованием XML DOM:

1. Методы для доступа к элементам:
   * **GetElementsByTagName(string name)** - возвращает коллекцию узлов с указанным именем тега.
   * **GetAttribute(string name)** - возвращает значение атрибута с указанным именем.
   * **GetElementsBуTagNameNS(string localName, string namespaceURI)** - возвращает коллекцию узлов с указанным локальным именем и пространством имен.
   * **SelectSingleNode(string xpath)** - выполняет запрос XPath и возвращает первый соответствующий узел.
2. Методы для создания и изменения элементов:
   * **CreateElement(string name)** - создает новый элемент с указанным именем.
   * **CreateAttribute(string name)** - создает новый атрибут с указанным именем.
   * **AppendChild(XmlNode newChild)** - добавляет указанный узел в конец списка дочерних узлов.
   * **InnerText** - свойство, позволяющее получить или установить текстовое содержимое узла.
   * **SetAttribute(string name, string value)** - устанавливает значение атрибута с указанным именем.
3. Методы для удаления элементов:
   * **RemoveChild(XmlNode oldChild)** - удаляет указанный узел из списка дочерних узлов.
   * **RemoveAttribute(string name)** - удаляет атрибут с указанным именем.

Пример создания XML-документа через класс XmlDocument:

using System;

using System.Xml;

class Program

{

static void Main()

{

// Создание XML-документа

XmlDocument xmlDoc = new XmlDocument();

// Создание корневого элемента

XmlElement rootElement = xmlDoc.CreateElement("Books");

xmlDoc.AppendChild(rootElement);

// Создание и добавление дочерних элементов

XmlElement bookElement1 = xmlDoc.CreateElement("Book");

XmlElement bookElement2 = xmlDoc.CreateElement("Book");

rootElement.AppendChild(bookElement1);

rootElement.AppendChild(bookElement2);

// Создание и установка атрибутов

XmlAttribute titleAttribute1 = xmlDoc.CreateAttribute("Title");

titleAttribute1.Value = "Book 1";

bookElement1.SetAttributeNode(titleAttribute1);

XmlAttribute titleAttribute2 = xmlDoc.CreateAttribute("Title");

titleAttribute2.Value = "Book 2";

bookElement2.SetAttributeNode(titleAttribute2);

// Сохранение XML-документа

xmlDoc.Save("books.xml");

Console.WriteLine("XML-документ успешно создан.");

Console.ReadLine();

}

}

1. Для чего предназначен класс System.Thread? Опишите основные функции для работы с потоком. Приведите пример использования нескольких потоков.

Класс **System.Thread** в C# предоставляет функциональность для работы с потоками. Поток представляет собой отдельный путь выполнения внутри программы, который может выполняться параллельно с другими потоками. Основные функции для работы с потоком включают:

1. Создание и запуск потока:
   * **Thread(ThreadStart start)** - конструктор, принимающий делегат **ThreadStart**, который указывает метод, который будет выполняться в потоке.
   * **Start()** - метод, запускающий поток и вызывающий метод, указанный в конструкторе.
   * **IsAlive** - свойство, возвращающее значение **true**, если поток находится в активном состоянии.
2. Управление выполнением потока:
   * **Sleep(int millisecondsTimeout)** - метод, приостанавливающий выполнение потока на указанное количество миллисекунд.
   * **Join()** - метод, блокирующий вызывающий поток до тех пор, пока не завершится целевой поток.
   * **Abort()** - метод, прерывающий выполнение потока.
3. Многопоточная синхронизация:
   * **Lock** - оператор, используемый для синхронизации доступа к общим данным между потоками.
   * **Monitor** - класс, предоставляющий методы для ожидания событий и синхронизации потоков.

Пример использования нескольких потоков:

using System;

using System.Threading;

class Program

{

static void Main()

{

// Создание и запуск потоков

Thread thread1 = new Thread(CountNumbers);

Thread thread2 = new Thread(PrintMessage);

thread1.Start(); // Запуск первого потока

thread2.Start(); // Запуск второго потока

// Ожидание завершения потоков

thread1.Join();

thread2.Join();

Console.WriteLine("Оба потока завершились.");

Console.ReadLine();

}

static void CountNumbers()

{

for (int i = 1; i <= 5; i++)

{

Console.WriteLine($"Поток 1: {i}");

Thread.Sleep(1000);

}

}

static void PrintMessage()

{

for (int i = 1; i <= 5; i++)

{

Console.WriteLine($"Поток 2: Привет, мир!");

Thread.Sleep(750);

}

}

}

1. Опишите структуру проекта WPF и XAML-кода. Перечислите основные типы контейнеров компоновки и элементов управления. Напишите программу, использующую контейнер Grid, элементы TextBox и Button для поиска самого длинного слова в строке.

Структура проекта WPF и XAML-кода:

Структура проекта WPF обычно включает в себя следующие элементы:

App.xaml - файл, содержащий глобальные ресурсы и настройки приложения.

MainWindow.xaml - основное окно приложения, определенное с использованием XAML.

MainWindow.xaml.cs - код-бехинд для главного окна, содержащий логику приложения.

Дополнительные XAML-файлы и код-бехинд для других окон или пользовательских элементов управления.

Ресурсы приложения, такие как изображения, стили, шрифты, сохраненные в каталоге проекта.

Папка ViewModel - содержит классы, представляющие модели представления (view model) для связывания данных.

XAML (eXtensible Application Markup Language) является декларативным языком разметки, который используется для определения пользовательского интерфейса в WPF. Он представляет собой XML-подобный синтаксис, который определяет структуру элементов управления и их свойства.

Основные типы контейнеров компоновки в WPF:

Grid - сетчатый контейнер, который размещает элементы в виде сетки из строк и столбцов.

StackPanel - контейнер, который располагает элементы последовательно в одном направлении (горизонтально или вертикально).

DockPanel - контейнер, который размещает элементы в одном из четырех направлений (вверху, внизу, слева или справа).

WrapPanel - контейнер, который автоматически переносит элементы на новую строку или столбец при достижении конца доступного пространства.

Основные элементы управления в WPF:

TextBox - текстовое поле для ввода или отображения текста.

Button - кнопка, которая реагирует на нажатие пользователем.

Label - метка для отображения текста или названия элемента управления.

ComboBox - комбинированный список для выбора одного значения из списка.

ListBox - список для отображения и выбора одного или нескольких элементов.

DataGrid - элемент для отображения и редактирования табличных данных.

Пример программы, использующей контейнер Grid, элементы TextBox и Button для поиска самого длинного слова в строке:

XAML (MainWindow.xaml):

xaml

Copy code

<Window x:Class="WpfApp.MainWindow"

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

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

Title="WPF App" Height="250" Width="400">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<Label Content="Введите строку:"/>

<TextBox x:Name="textBox" Grid.Row="1" Margin="5"/>

<Button Content="Найти самое длинное слово" Grid.Row="2" Margin="5" Click="Button\_Click"/>

</Grid>

</Window>

C# (MainWindow.xaml.cs):

csharp

Copy code

using System.Linq;

using System.Windows;

namespace WpfApp

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

string input = textBox.Text;

string[] words = input.Split(' ');

string longestWord = words.OrderByDescending(w => w.Length).FirstOrDefault();

MessageBox.Show($"Самое длинное слово: {longestWord}");

}

}

}

В этом примере при нажатии на кнопку "Найти самое длинное слово" программа разделяет введенную строку на отдельные слова и находит самое длинное из них. Результат выводится в сообщении MessageBox. Контейнер Grid используется для размещения метки, текстового поля и кнопки в вертикальной сетке.

1. Опишите процесс привязки базы данных к проекту WPF с помощью технологии ADO.NET или Entity Framework.

Привязка базы данных к проекту WPF может быть осуществлена с помощью технологий ADO.NET или Entity Framework. Вот описание процесса для обоих вариантов:

Привязка базы данных с использованием ADO.NET:

Создайте подключение к базе данных с помощью класса SqlConnection, указав строку подключения, содержащую информацию о сервере базы данных, учетных данных и других настройках.

Используйте SqlCommand для выполнения запросов к базе данных, таких как выборка данных или изменение записей.

Получите результаты запроса и свяжите их с элементами управления WPF для отображения данных.

Обработайте события элементов управления WPF, чтобы обновить данные в базе данных при необходимости.

Привязка базы данных с использованием Entity Framework:

Установите Entity Framework через NuGet Package Manager в вашем проекте.

Создайте модель данных (Entity Data Model) с помощью инструментов Entity Framework, которые позволят вам создать классы, представляющие таблицы базы данных и их связи.

Используйте контекст данных (DbContext) для выполнения операций с базой данных, таких как выборка, вставка, обновление и удаление записей.

Привяжите объекты данных Entity Framework к элементам управления WPF с помощью привязки данных (data binding) для автоматического отображения и обновления данных.

Обработайте события элементов управления WPF и измените соответствующие объекты данных Entity Framework для сохранения изменений в базе данных.

Пример привязки базы данных с использованием Entity Framework:

Установите Entity Framework через NuGet Package Manager.

Создайте классы, представляющие таблицы базы данных (например, Product и Category):

csharp

Copy code

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public int CategoryId { get; set; }

public Category Category { get; set; }

}

public class Category

{

public int Id { get; set; }

public string Name { get; set; }

}

Создайте класс контекста данных, наследующийся от DbContext:

public class MyDbContext : DbContext

{

public DbSet<Product> Products { get; set; }

public DbSet<Category> Categories { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("your\_connection\_string");

}

}

В XAML определите элементы управления, которые будут отображать данные (например, DataGrid):

xaml

Copy code

<DataGrid ItemsSource="{Binding Products}" AutoGenerateColumns="False">

<DataGrid.Columns>

<DataGridTextColumn Header="Name" Binding="{Binding Name}" />

<DataGridTextColumn Header="Price" Binding="{Binding Price}" />

</DataGrid.Columns>

</DataGrid>

В коде C# свяжите данные с элементами управления и выполните запрос к базе данных:

public partial class MainWindow : Window

{

private MyDbContext dbContext;

public MainWindow()

{

InitializeComponent();

dbContext = new MyDbContext();

DataContext = dbContext;

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

// Выполните необходимые операции с базой данных, используя dbContext

var products = dbContext.Products.ToList();

}

}

В этом примере данные из таблицы Products связываются с элементом DataGrid, и при нажатии кнопки данные выбираются из базы данных с использованием контекста данных dbContext.

Обратите внимание, что для полноценного использования Entity Framework и ADO.NET в проекте WPF требуется подготовка и настройка соответствующей базы данных, настройка подключения и правильное использование соответствующих классов и методов в коде.

1. Опишите структуру паттерна MVVM. Укажите и поясните составляющие паттерна MVVM. Приведите пример реализации этого паттерна на C#.

MVVM (Model-View-ViewModel) - это паттерн проектирования, используемый в разработке программного обеспечения, особенно в приложениях с графическим интерфейсом пользователя, таких как WPF и Silverlight. Он разделяет компоненты приложения на три основных части: модель (Model), представление (View) и модель представления (ViewModel), что обеспечивает разделение ответственностей и упрощает поддержку и тестирование приложения.

Структура паттерна MVVM включает в себя следующие составляющие:

Модель (Model):

Представляет бизнес-логику и данные приложения.

Обычно содержит классы, которые представляют объекты и операции с данными.

Не зависит от представления или модели представления.

Представление (View):

Отображает данные модели пользователю.

Отвечает за визуализацию элементов управления и отображение данных.

Обычно реализуется с использованием XAML.

Модель представления (ViewModel):

Служит посредником между моделью и представлением.

Предоставляет данные и команды, необходимые для отображения и взаимодействия с пользователем.

Реализует логику, связанную с представлением.

Осуществляет привязку данных между моделью и представлением с использованием привязки данных (data binding).

Пример реализации паттерна MVVM на C#:

Модель (Model):

public class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

Модель представления (ViewModel):

public class PersonViewModel : INotifyPropertyChanged

{

private Person person;

public string FirstName

{

get { return person.FirstName; }

set

{

if (person.FirstName != value)

{

person.FirstName = value;

OnPropertyChanged();

}

}

}

public string LastName

{

get { return person.LastName; }

set

{

if (person.LastName != value)

{

person.LastName = value;

OnPropertyChanged();

}

}

}

public PersonViewModel()

{

person = new Person();

}

public event PropertyChangedEventHandler PropertyChanged;

protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

xaml

Copy code

<Window x:Class="WpfApp.MainWindow"

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

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

Title="WPF App" Height="250" Width="400">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<Label Content="First Name:"/>

<TextBox Grid.Row="0" Grid.Column="1" Text="{Binding FirstName, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"/>

<Label Content="Last Name:" Grid.Row="1"/>

<TextBox Grid.Row="1" Grid.Column="1" Text="{Binding LastName, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}"/>

<Button Grid.Row="2" Content="Submit" Click="Button\_Click"/>

</Grid>

</Window>

В коде C# свяжите представление с моделью представления:

public partial class MainWindow : Window

{

private PersonViewModel personViewModel;

public MainWindow()

{

InitializeComponent();

personViewModel = new PersonViewModel();

DataContext = personViewModel;

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

// Обработка события нажатия кнопки

MessageBox.Show($"Hello, {personViewModel.FirstName} {personViewModel.LastName}!");

}

}

В этом примере при изменении текста в текстовых полях представления, связанных с FirstName и LastName, данные автоматически обновляются в модели представления благодаря привязке данных. При нажатии кнопки данные из модели представления выводятся в сообщении MessageBox.

Преподаватели В.Ю.Михалевич

Е.В.Багласова

А.В.Кривошеина

Е.Н.Коропа

Рассмотрено на заседании цикловой комиссии Программного обеспечения информационных технологий №10

Протокол №\_\_\_от «\_\_»\_\_\_\_\_\_\_20\_\_

Председатель ЦК\_\_\_\_\_\_\_\_\_\_В.Ю.Михалевич