Вопросы к зачету по дисциплинам:

«Объектно-ориентированные технологии программирования и стандарты проектирования»

«Объектно-ориентированное проектирование и программирование»

1. Состав. NET Framework. Структура среды выполнения CLR.

**Состав .NET Framework**

**.NET Framework** — это платформа разработки от Microsoft, предоставляющая интегрированную среду для создания, выполнения и управления приложениями на основе .NET. Основные компоненты:

1. **Common Language Runtime (CLR)** — среда выполнения, управляющая выполнением кода.
2. **Библиотека классов (Base Class Library, BCL)** — стандартная библиотека, содержащая классы и методы для выполнения базовых операций (работа с файлами, потоками, сетью и т. д.).
3. **ADO.NET** — технологии для работы с базами данных.
4. **ASP.NET** — платформа для создания веб-приложений.
5. **Windows Forms (WinForms)** — API для создания десктопных приложений с графическим интерфейсом.
6. **WPF (Windows Presentation Foundation)** — для создания более современных и визуально сложных десктопных приложений.
7. **WCF (Windows Communication Foundation)** — для создания распределенных систем и взаимодействия через сети.
8. **LINQ (Language Integrated Query)** — технология для работы с данными в виде запросов.
9. **Компиляторы и инструменты** — поддержка языков (C#, VB.NET, F#) и разработка в средах типа Visual Studio.

**Структура среды выполнения CLR**

**Common Language Runtime (CLR)** — ядро .NET Framework, выполняющее следующие ключевые функции:

**1. Загрузка и управление кодом**

* **MSIL (Microsoft Intermediate Language):** Код компилируется в промежуточный язык MSIL, который исполняется CLR.
* **JIT-компиляция (Just-In-Time):** В процессе выполнения MSIL компилируется в машинный код для конкретной платформы.

**2. Управление памятью**

* **Сборка мусора (Garbage Collection, GC):** CLR автоматически управляет памятью, очищая объекты, которые больше не используются.
* **Управляемая куча:** Память выделяется и освобождается с использованием управляемой кучи.

**3. Обеспечение безопасности**

* **CAS (Code Access Security):** Ограничение доступа к ресурсам на основе политики безопасности.
* **Верификация кода:** Проверка кода на корректность перед выполнением.

**4. Управление потоками**

* Поддержка многопоточности и управление синхронизацией потоков.

**5. Обработка исключений**

* Центральный механизм обработки исключений для всех поддерживаемых языков.

**6. Интеграция языков**

* Позволяет использовать компоненты, написанные на разных языках .NET (C#, VB.NET, F#, и др.), благодаря поддержке стандарта **Common Language Specification (CLS)**.

**7. Интероперабельность**

* **P/Invoke:** Вызов функций из нативных библиотек.
* **COM Interop:** Взаимодействие с компонентами COM.

1. Структура управляемого модуля - portable executable (PE). Понятие и исполнение сборки. CIL.

**Структура управляемого модуля (PE) в .NET Framework**

**Portable Executable (PE)** — это формат файла, используемый для хранения исполняемого кода или библиотек в Windows. В контексте .NET, PE-модуль содержит не только машинный код, но и дополнительные структуры для исполнения кода в управляемой среде Common Language Runtime (CLR). Такие модули называются **управляемыми PE-модулями**.

**Основные компоненты PE-файла:**

1. **Заголовок PE:**
   * Стандартный заголовок Windows PE, используемый всеми Windows-приложениями.
   * Указывает операционной системе, как загрузить файл.
2. **Метаданные:**
   * Информация, описывающая сборку, её типы, методы, свойства, события и другие элементы.
   * Содержит таблицы, которые используются для работы с кодом в CLR.
   * Примеры метаданных:
     + Информация о версиях.
     + Имя сборки.
     + Список зависимых библиотек.
3. **Код CIL (Common Intermediate Language):**
   * Промежуточный язык, в который компилируется исходный код на C#.
   * Независим от платформы и выполняется в CLR.
   * Содержит инструкции высокого уровня, которые позже преобразуются в машинный код (JIT-компиляция).
4. **Кэшированный машинный код (опционально):**
   * Может содержать предварительно скомпилированный машинный код для ускорения выполнения (например, через NGen).
5. **Ресурсы:**
   * Встроенные файлы, такие как изображения, строки, XML, локализация и т.д.

**Исполнение сборки в .NET**

Сборка — это единица развертывания, выполнения и управления версиями в .NET. Она может быть представлена одним или несколькими файлами. Основные шаги исполнения:

1. **Загрузка сборки:**
   * CLR загружает PE-файл в память с помощью **assembly loader**.
   * Проверяются метаданные и зависимости.
2. **JIT-компиляция:**
   * Код CIL компилируется в машинный код во время выполнения.
   * Используется **Just-In-Time (JIT)** компилятор.
3. **Исполнение кода:**
   * CLR управляет выполнением, предоставляя такие функции, как:
     + Управление памятью (Garbage Collection).
     + Обеспечение безопасности типов.
     + Обработка исключений.

**CIL (Common Intermediate Language)**

CIL — это язык промежуточного представления, используемый для платформы .NET. Ранее назывался MSIL (Microsoft Intermediate Language).

**Особенности CIL:**

* Независимость от платформы.
* Инструкции высокого уровня (например, работа с объектами, вызовы методов).
* Используется для обеспечения совместимости между языками .NET (C#, VB.NET, F# и др.).

**Пример CIL-кода:**

.method public hidebysig static void Main() cil managed

{

.entrypoint

ldstr "Hello, World!" // Загрузка строки "Hello, World!"

call void [mscorlib]System.Console::WriteLine(string)

ret // Возврат из метода

}

**Исполнение CIL в CLR**

1. **Загрузка сборки:** CLR считывает метаданные и загружает нужные типы.
2. **JIT-компиляция:** Код CIL транслируется в машинный код, оптимизированный для конкретной платформы.
3. **Выполнение:** CLR управляет выполнением программы, обеспечивая:
   * Безопасность типов.
   * Управление памятью.
   * Поддержку многопоточности.

**Основные понятия о сборках в .NET**

* **Простая сборка:** Один файл (обычно .exe или .dll), содержащий CIL и метаданные.
* **Многомодульная сборка:** Сборка состоит из нескольких файлов.
* **Сильное имя:** Сборка может быть подписана с использованием криптографического ключа для идентификации.
* **Глобальный кэш сборок (GAC):** Хранилище для сборок, которые могут использоваться разными приложениями.

**Итог**

Структура PE и использование CIL позволяют платформе .NET быть независимой от операционной системы и процессора, обеспечивать безопасность кода и эффективное управление памятью. Благодаря JIT-компиляции код оптимизируется для конкретной машины, что увеличивает производительность программ.

3. CTS (Common Type System). Типы данных C#. Ссылочные и типы значений.

**ChatGPT сказал:**

ChatGPT

**Common Type System (CTS) в .NET**

**Common Type System (CTS)** — это часть платформы .NET, которая определяет правила объявления, использования и управления типами данных. CTS обеспечивает совместимость между различными языками, поддерживаемыми .NET, такими как C#, VB.NET, F# и другими.

**Основные аспекты CTS**

1. **Единый набор типов данных:**
   * CTS стандартизирует типы данных, чтобы они были понятны и совместимы между языками .NET.
   * Например, int в C# соответствует System.Int32 в CTS.
2. **Поддержка объектной модели:**
   * Все типы в CTS являются производными от базового класса System.Object.
3. **Два вида типов:**
   * **Значимые типы (Value Types):** Хранят данные непосредственно.
   * **Ссылочные типы (Reference Types):** Хранят ссылку на область памяти, где находятся данные.

**Типы данных в C#**

C# предоставляет типы данных, которые делятся на **примитивные**, **сложные** и **пользовательские**.

**1. Примитивные типы (built-in types):**

| **Тип** | **CTS тип** | **Диапазон значений** | **Пример значения** |
| --- | --- | --- | --- |
| byte | System.Byte | 0 до 255 | 100 |
| int | System.Int32 | -2,147,483,648 до 2,147,483,647 | -10 |
| long | System.Int64 | -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807 | 1234567890 |
| float | System.Single | ±1.5 × 10⁻⁴⁵ до ±3.4 × 10³⁸ | 3.14f |
| double | System.Double | ±5.0 × 10⁻³²⁴ до ±1.7 × 10³⁰⁸ | 3.14159 |
| char | System.Char | Один символ Unicode | 'A' |
| bool | System.Boolean | true или false | true |

**2. Ссылочные типы:**

| **Тип** | **CTS тип** | **Пример** |
| --- | --- | --- |
| object | System.Object | Хранит любой тип данных. |
| string | System.String | Последовательность символов. |
| dynamic | System.Object | Определяется во время выполнения. |

**Сравнение значимых и ссылочных типов**

| **Характеристика** | **Значимые типы (Value Types)** | **Ссылочные типы (Reference Types)** |
| --- | --- | --- |
| **Хранение данных** | Данные хранятся в стеке | Ссылка на данные хранится в стеке, а данные — в куче. |
| **Копирование** | Копируется само значение | Копируется только ссылка. |
| **Пример** | int, float, bool | string, object, class |
| **Базовый класс** | System.ValueType | System.Object |
| **Производительность** | Быстрее из-за хранения в стеке | Медленнее из-за операций в куче |

1. Понятие упаковки и распаковки типов. Типы Nullable: преобразование, проверка, null-объединение

**Понятие упаковки и распаковки типов (Boxing и Unboxing)**

Упаковка (**Boxing**) и распаковка (**Unboxing**) — это процессы преобразования значимых типов (Value Types) в ссылочные типы (Reference Types) и наоборот в .NET.

**Упаковка (Boxing):**

* Преобразование значимого типа (например, int, double) в ссылочный тип (object).
* Данные значимого типа упаковываются в объект, размещаемый в управляемой куче (Heap).
* Это происходит неявно.

**Пример упаковки:**

int num = 42; // Значимый тип

object obj = num; // Упаковка (Boxing)

**Распаковка (Unboxing):**

* Обратный процесс: преобразование ссылочного типа (object) обратно в значимый тип.
* Требуется явное приведение типа.
* Если тип не совпадает, возникает исключение InvalidCastException.

**Пример распаковки:**

object obj = 42; // Упаковка

int num = (int)obj; // Распаковка (Unboxing)

**Nullable-типы**

**Что такое Nullable?**

Nullable-типы предоставляют возможность значимым типам принимать значение null. Это удобно, когда тип данных должен поддерживать отсутствие значения, как у ссылочных типов.

В .NET Nullable-тип определяется как Nullable<T>, где T — это любой значимый тип. В C# для удобства используется синтаксис T?.

**Пример:**

int? nullableInt = null; // Значимый тип int, который может быть null

**Операции с Nullable-типа**

1. **Проверка на значение:**
   * Используйте свойство .HasValue, чтобы узнать, имеет ли Nullable-переменная значение.
   * Свойство .Value возвращает значение переменной, если оно есть. Если значение отсутствует, возникает исключение.

**Пример:**

int? x = 10;

if (x.HasValue)

{

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

}

else

{

Console.WriteLine("Значение отсутствует (null).");

}

1. **Установка значения:**
   * Вы можете присвоить null или конкретное значение.
2. int? y = null;
3. y = 20; // Теперь переменная имеет значение
4. **Null-объединение (??):**
   * Используется для задания значения по умолчанию, если Nullable-переменная равна null.

**Пример:**

int? z = null;

int defaultValue = z ?? -1; // Если z == null, то значение -1

Console.WriteLine(defaultValue); // Вывод: -1

1. **Null-условный оператор (?.):**
   * Позволяет вызывать методы или свойства объекта, только если он не равен null.

**Пример:**

string? text = null;

Console.WriteLine(text?.Length ?? 0); // Вывод: 0

**Nullable и преобразование**

1. **Неявное преобразование Nullable-типа в его базовый тип:**
   * Если у Nullable-типа есть значение, то преобразование извлекает это значение.
2. int? nullableInt = 42;
3. int regularInt = nullableInt ?? 0; // Если null, то используется значение 0
4. **Явное преобразование базового типа в Nullable-тип:**
   * Базовый тип можно преобразовать в Nullable через неявное приведение.
5. int value = 50;
6. int? nullableValue = value;
7. **Сравнение с null:**
   * Nullable-типы можно сравнивать с null напрямую.
8. int? a = null;
9. Console.WriteLine(a == null); // true

**Преимущества использования Nullable-типа:**

1. Позволяет значимым типам быть более гибкими.
2. Уменьшает вероятность ошибок, связанных с отсутствием значения.
3. Улучшает читаемость и устойчивость кода через использование ?? и ?..

**Итоги**

* **Упаковка и распаковка** позволяют работать с значимыми типами как со ссылочными, но они добавляют накладные расходы, поэтому их следует избегать, где это возможно.
* **Nullable-типы** позволяют значимым типам принимать null, что делает их более универсальными.
* Операторы ?? и ?. облегчают обработку случаев, связанных с null.

5. Тип данных String: операции, литералы, пустые и нулевые строки, форматированный вывод.

**Тип данных String в C#**

Тип string в C# представляет последовательность символов Unicode. Он является ссылочным типом и неизменяемым (immutable), то есть любые операции над строкой создают новую строку.

**Операции со строками**

1. **Конкатенация:** Объединение строк с помощью оператора + или метода string.Concat.
2. string s1 = "Hello, ";
3. string s2 = "world!";
4. string result = s1 + s2; // "Hello, world!"
5. **Извлечение подстроки:** Используйте метод Substring.
6. string s = "Hello, world!";
7. string part = s.Substring(7, 5); // "world"
8. **Поиск символов или строк:** Методы IndexOf, LastIndexOf, Contains.
9. bool contains = s.Contains("world"); // true
10. int index = s.IndexOf('o'); // 4
11. **Изменение регистра:** Методы ToUpper, ToLower.
12. string upper = s.ToUpper(); // "HELLO, WORLD!"
13. **Разбиение строки:** Метод Split.
14. string[] words = s.Split(' '); // ["Hello,", "world!"]
15. **Удаление пробелов:** Методы Trim, TrimStart, TrimEnd.
16. string trimmed = " Hello ".Trim(); // "Hello"
17. **Сравнение строк:** Методы Equals, Compare, CompareOrdinal.
18. bool isEqual = s1.Equals(s2, StringComparison.OrdinalIgnoreCase); // сравнение без учета регистра

**Литералы строк**

1. **Обычные строки:** Заключаются в двойные кавычки.
2. string s = "Hello, world!";
3. **Строки с экранированием:** Используются символы \ для спецсимволов.
4. string s = "Line1\nLine2";
5. **Дословные строки (verbatim):** Начинаются с символа @. Игнорируют символы экранирования.
6. string path = @"C:\Users\Documents\File.txt";
7. **Интерполяция строк:** Используется $ перед строкой для вставки значений.
8. int age = 25;
9. string s = $"I am {age} years old.";

**Пустые и нулевые строки**

1. **Пустая строка (""):** Строка, которая содержит 0 символов.
2. string empty = "";
3. bool isEmpty = string.IsNullOrEmpty(empty); // true
4. **Нулевая строка (null):** Отсутствие строки.
5. string nullString = null;
6. bool isNull = nullString == null; // true
7. **Методы для проверки:**
   * string.IsNullOrEmpty: проверяет, является ли строка пустой или null.
   * string.IsNullOrWhiteSpace: проверяет, состоит ли строка только из пробелов или является пустой/null.

**Форматированный вывод**

1. **Метод String.Format:**
2. string formatted = string.Format("Name: {0}, Age: {1}", "Alice", 30);
3. **Интерполяция строк:** Более удобная альтернатива String.Format.
4. string name = "Alice";
5. int age = 30;
6. string result = $"Name: {name}, Age: {age}";
7. **Форматирование чисел и дат:**
8. double price = 123.456;
9. string formatted = $"Price: {price:F2}"; // "Price: 123.46"
10. DateTime now = DateTime.Now;
11. string dateFormatted = $"Today: {now:yyyy-MM-dd}";

**Итоги**

* string — мощный тип для работы с текстами.
* Основные методы позволяют выполнять поиск, извлечение, изменение, форматирование и сравнение строк.
* Различие между пустыми и нулевыми строками важно для предотвращения ошибок.
* Используйте интерполяцию ($"...") для удобного форматирования.

6. Неявная типизация – назначение и использование.

**Неявная типизация в C#**

Неявная типизация позволяет компилятору автоматически определять тип переменной на основе присваиваемого значения. Это делается с помощью ключевого слова var.

**Назначение:**

1. **Упрощение кода:** Сокращает явное указание типов, делая код более читаемым.
2. var number = 10; // компилятор автоматически определяет тип как int
3. **Гибкость:** Удобно использовать с анонимными типами и LINQ-запросами, где тип данных заранее неизвестен.
4. var person = new { Name = "Alice", Age = 30 }; // анонимный тип
5. **Сокращение избыточности:** Избавляет от необходимости явно указывать длинные имена типов.
6. var dictionary = new Dictionary<int, List<string>>();

**Использование:**

1. **Объявление переменной:** Переменная объявляется с помощью var, но ее тип фиксируется компилятором при первом присваивании.
2. var text = "Hello"; // тип string
3. var isActive = true; // тип bool
4. **Сложные типы:** Используется для сокращения записи сложных типов.
5. var list = new List<int> { 1, 2, 3 }; // тип List<int>
6. **LINQ-запросы:** Неявная типизация необходима для работы с результатами запросов.
7. var result = myCollection.Where(x => x > 10);

**Ограничения:**

1. **Необходимо сразу инициализировать:** Нельзя объявить var без присвоения значения.
2. var x; // Ошибка!
3. **Тип нельзя изменить:** После присвоения тип переменной становится неизменяемым.
4. var number = 10;
5. number = "text"; // Ошибка!
6. **Не используется для динамических данных:** Для данных с неопределенным типом используйте dynamic.

**Итоги:**

* Используйте var для удобства, но не злоупотребляйте, чтобы код оставался понятным.
* Применяйте его в случаях, когда тип очевиден из контекста или сложно указать явно (например, при работе с LINQ и анонимными типами).

7. Массивы C# одномерные, прямоугольные и ступенчатые.

**Массивы в C#**

Массивы — это коллекция элементов одного типа, размещенных в памяти последовательно.

**Типы массивов в C#:**

1. **Одномерные массивы:**
   * Самый простой вид массивов, представляющий собой линейный список элементов.
   * Объявление:
   * int[] array = new int[5]; // массив из 5 элементов
   * array[0] = 10; // присвоение значения
   * Инициализация:
   * int[] array = { 1, 2, 3, 4, 5 };
2. **Прямоугольные массивы (многомерные):**
   * Это массивы, в которых каждая строка содержит одинаковое количество элементов.
   * Объявление:
   * int[,] matrix = new int[3, 4]; // массив 3x4
   * Инициализация:
   * int[,] matrix = {
   * { 1, 2, 3, 4 },
   * { 5, 6, 7, 8 },
   * { 9, 10, 11, 12 }
   * };
   * Обращение к элементу:
   * int value = matrix[1, 2]; // элемент второй строки и третьего столбца
3. **Ступенчатые массивы (jagged arrays):**
   * Это массивы массивов, где строки могут иметь разное количество элементов.
   * Объявление:
   * int[][] jaggedArray = new int[3][]; // массив из 3 массивов
   * jaggedArray[0] = new int[] { 1, 2, 3 };
   * jaggedArray[1] = new int[] { 4, 5 };
   * jaggedArray[2] = new int[] { 6, 7, 8, 9 };
   * Обращение к элементу:
   * int value = jaggedArray[1][0]; // первый элемент второй строки

**Общие операции с массивами:**

* Получение длины:
* int length = array.Length;
* Перебор элементов:
* foreach (int item in array)
* {
* Console.WriteLine(item);
* }
* Копирование:
* int[] newArray = new int[5];
* Array.Copy(array, newArray, 5);

**Итоги:**

* **Одномерные массивы:** подходят для линейных списков.
* **Прямоугольные массивы:** удобны для работы с матрицами и таблицами.
* **Ступенчатые массивы:** полезны, если строки имеют разную длину.

8. Понятие кортежей. Свойства, создание

**Кортежи в C#**

Кортежи (Tuples) — это структура данных, позволяющая объединять несколько значений разного типа в один объект.

**Свойства кортежей:**

1. **Могут содержать элементы разных типов.**
2. var tuple = (1, "Hello", 3.5);
   * Первый элемент: int
   * Второй элемент: string
   * Третий элемент: double
3. **Имеют ограниченное количество элементов (до 8).**
   * Для большего количества используется вложение.
4. var largeTuple = (1, 2, 3, 4, 5, 6, 7, (8, 9));
5. **Являются неизменяемыми.**
   * Значения элементов кортежа нельзя изменить после создания.
6. **Могут быть именованными.**
7. var namedTuple = (Id: 1, Name: "Alice", Age: 30);
8. Console.WriteLine(namedTuple.Name); // Вывод: Alice

**Создание кортежей:**

1. **Неименованные элементы:**
2. var tuple = (10, "World", true);
3. Console.WriteLine(tuple.Item1); // 10
4. Console.WriteLine(tuple.Item2); // World
5. Console.WriteLine(tuple.Item3); // True
6. **Именованные элементы:**
7. var person = (Id: 1, Name: "Bob", Age: 25);
8. Console.WriteLine(person.Name); // Bob
9. **Использование в методах:**
   * **Возврат кортежа из метода:**
   * (int, string) GetPerson()
   * {
   * return (1, "John");
   * }
   * var person = GetPerson();
   * Console.WriteLine(person.Item2); // John
   * **Декомпозиция:**
   * var (id, name) = GetPerson();
   * Console.WriteLine(name); // John
10. **Применение с LINQ:**
11. var data = collection.Select(x => (x.Id, x.Name));

**Использование кортежей:**

* Удобны для передачи нескольких значений из метода.
* Полезны для временного хранения данных без создания отдельного класса.

**Итоги:**

* **Простота:** Легко создавать и использовать.
* **Удобство:** Позволяют возвращать несколько значений из методов.
* **Гибкость:** Поддерживают именованные элементы для улучшения читаемости.

9. Принципы объектно-ориентированного программирования.

**Принципы объектно-ориентированного программирования (ООП)**

Объектно-ориентированное программирование основано на четырёх ключевых принципах: **инкапсуляция, наследование, полиморфизм** и **абстракция**. Они обеспечивают модульность, повторное использование кода и удобство разработки.

**1. Инкапсуляция**

* Сокрытие внутренней реализации объекта и предоставление доступа только через публичные методы.
* **Цель:** Уменьшение зависимости кода от деталей реализации.
* **Пример:**
* class Person
* {
* private string name; // Закрытое поле
* public string Name // Свойство для доступа
* {
* get { return name; }
* set { name = value; }
* }
* }

**2. Наследование**

* Возможность создавать новые классы на основе существующих.
* **Цель:** Повторное использование кода и создание иерархий.
* **Пример:**
* class Animal
* {
* public void Eat() => Console.WriteLine("Eating...");
* }
* class Dog : Animal
* {
* public void Bark() => Console.WriteLine("Barking...");
* }

**3. Полиморфизм**

* Способность объекта принимать множество форм.
* **Цель:** Упрощение обработки объектов, относящихся к одному базовому классу.
* **Пример:**
* class Shape
* {
* public virtual void Draw() => Console.WriteLine("Drawing Shape");
* }
* class Circle : Shape
* {
* public override void Draw() => Console.WriteLine("Drawing Circle");
* }
* class Rectangle : Shape
* {
* public override void Draw() => Console.WriteLine("Drawing Rectangle");
* }
* void Render(Shape shape) => shape.Draw();

**4. Абстракция**

* Сокрытие деталей реализации и предоставление только существенного интерфейса.
* **Цель:** Упрощение работы с объектами и обеспечение гибкости.
* **Пример:**
* abstract class Animal
* {
* public abstract void MakeSound(); // Только описание
* }
* class Dog : Animal
* {
* public override void MakeSound() => Console.WriteLine("Bark");
* }

**Итоги:**

* **Инкапсуляция:** Упрощает управление данными и защищает их.
* **Наследование:** Позволяет расширять существующий функционал.
* **Полиморфизм:** Унифицирует обработку объектов.
* **Абстракция:** Скрывает сложные детали, оставляя только важное.

10. Класс. Элементы класса. Свойства и индексаторы.

**Класс в C#**

Класс в C# — это основная структура, которая описывает тип данных, определяя поля, методы, свойства и другие члены. Классы являются основой объектно-ориентированного программирования.

**Элементы класса:**

1. **Поля (Fields):**
   * Переменные, хранящие данные.
   * Обычно имеют модификатор доступа (private, public и т.д.).
2. class Person
3. {
4. private string name;
5. }
6. **Методы (Methods):**
   * Функции, выполняющие действия с данными.
7. class Person
8. {
9. public void Greet()
10. {
11. Console.WriteLine("Hello, " + name);
12. }
13. }
14. **Свойства (Properties):**
    * Специальные методы для получения и установки значений полей.
    * Представляют собой сочетание методов get и set.
15. class Person
16. {
17. private string name;
18. public string Name // Свойство
19. {
20. get { return name; }
21. set { name = value; }
22. }
23. }
24. **Конструкторы (Constructors):**
    * Методы, вызываемые при создании объекта для инициализации его состояния.
25. class Person
26. {
27. public string Name { get; set; }
28. public Person(string name) // Конструктор
29. {
30. Name = name;
31. }
32. }
33. **События (Events) и Делегаты (Delegates):**
    * Используются для обработки событий, например, при нажатии кнопки или изменении состояния.
34. public event EventHandler NameChanged;
35. **Индексаторы (Indexers):**
    * Позволяют обращаться к объектам класса как к массивам.
    * Используются для предоставления доступа к элементам через индекс.
36. class MyCollection
37. {
38. private string[] items = new string[10];
39. public string this[int index] // Индексатор
40. {
41. get { return items[index]; }
42. set { items[index] = value; }
43. }
44. }

**Свойства:**

* Свойства предоставляют механизм для инкапсуляции данных.
* Они могут быть простыми или иметь дополнительные блоки get и set, которые можно настроить.
* Пример: Свойство с логикой проверки.
* public int Age
* {
* get { return age; }
* set { if (value >= 0) age = value; }
* }

**Индексаторы:**

* Позволяют объектам класса быть индексируемыми, как массивы.
* В отличие от методов, индексаторы используют ключевое слово this и могут принимать параметры (например, индексы).
* public string this[int index]
* {
* get { return items[index]; }
* set { items[index] = value; }
* }

**Пример класса с элементами:**

class Person

{

private string name;

private int age;

public string Name // Свойство

{

get { return name; }

set { name = value; }

}

public int Age // Свойство

{

get { return age; }

set { age = value; }

}

public Person(string name, int age) // Конструктор

{

Name = name;

Age = age;

}

public void Greet() // Метод

{

Console.WriteLine($"Hello, {Name}. You are {Age} years old.");

}

}

**Итоги:**

* **Классы** — это шаблоны для создания объектов.
* **Свойства** позволяют контролировать доступ к данным объекта.
* **Индексаторы** позволяют использовать объекты как массивы.
* **Методы** реализуют поведение объектов.

11. Класс. Константы. Поля только для чтения. Инициализаторы класса.

12. Спецификаторы доступа C#. Видимость типов. Доступ к членам типов.

13. Класс. Конструкторы и их свойства. Деструкторы

14. Класс и методы System.Object.

15. Статические методы и статические конструкторы класса.

16. Статические классы. Методы расширения и правила их определения.

17. Анонимные типы.

18. Модификаторы параметров - ref , out, params. Необязательные и именованные аргументы.

19. Перегрузка методов и операторов. Правила перегрузки операторов.

20. Операции преобразования типа. Явная и неявная форма. Ограничения.

21. Вложенные типы. Вложенные объекты

22. Правила наследования C#.

23. Сокрытие имен при наследовании. Обращение к срытым членам

24. Использование операций is и as

25. Полиморфизм. Виртуальные методы, свойства и индексаторы. Правила переопределения.

26. Понятие раннего и позднего связывания.

27. Абстрактные классы и методы. Бесплодные классы.

28. Структуры в C#.

29. Интерфейсы. Свойства интерфейсов. Реализация интерфейсов.

30. Явная и неявная реализация интерфейсов. Работа с объектами через интерфейсы.

31. Ковариантность интерфейсов. Контравариантность интерфейсов

32. Стандартные интерфейсы .NET. Назначение и применение.

33. Исключительные ситуации. Генерация и повторная генерация исключений.

34. Исключительные ситуации. Варианты обработки исключений. Фильтры исключений

35. Обобщения (generics). Свойства обобщений.

36. Концепция ограничений обобщений. Статические члены обобщений.

37. Делегаты. Определение, назначение и варианты использования. Обобщенные делегаты.

38. Анонимные функции. Лямбда-выражения.

39. Обобщённые делегаты .NET. Action, Func, Predicate

40. События и делегаты.

41. Стандартные коллекции .NET. Типы коллекций.

42. Стандартные интерфесы коллекций.

43. IEnumerable и IEnumerator

44. LINQ to Objects. Синтаксис. Форма. Возврат результата. Грамматика выражений запросов. Отложенные и неотлаженные операции.

45. LINQ to Objects. Операции Where, Select, Take, OrderB, Join, GroupBy

46. Рефлексия. System Type.

47. Классы для работы с файловой системой.

48. Синтаксическая конструкция using. Чтение и запись файлов. Потоковые классы.

49. Классы адаптеры потоков.

50. Сериализация. Форматы сериализации.

51. Сериализация контрактов данных. интерфейс ISerializable.

52. Атрибуты. Создание собственного атрибута.

53. Процесс. Домен приложений. Поток выполнения.

54. Создание потоков , классы приоритетов. Состояния потоков

55. Синхронизация потоков. Lock. Monitor. Мutex. Semaphore

56. Библиотека параллельных задач TPL. Класс Task. Состояние задачи.

57. Способы создания Task. Возврат результата. Отмена выполнения задач. Продолжения.

58. Параллелизм при императивной обработке данных. Класс Parallel

59. Асинхронные методы. async и await

60. Проектирование отношений. Агрегация, композиция и ассоциация

61. Антипаттерны проектирования. Рефакторинг. Методы рефакторинга.

62. Чистый код. Требования к именам, функциям, форматированию.

63. Чистый код. Требования к классам и объектам.