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

.NET Framework представляет собой платформу разработки приложений, созданную компанией Microsoft. Она включает в себя несколько ключевых компонентов, в том числе Common Language Runtime (CLR), библиотеки классов (Class Libraries), и ASP.NET (для веб-приложений). Давайте рассмотрим структуру и особенности CLR.

1. **Common Language Runtime (CLR):** CLR является основным компонентом .NET Framework. Его задача - предоставить среду выполнения для управляемых приложений. Вот ключевые элементы CLR:
   * **Just-In-Time Compiler (JIT):** Код на управляемых языках, таких как C#, не компилируется напрямую в машинный код. Вместо этого он компилируется в промежуточный язык MSIL (Microsoft Intermediate Language), который затем JIT-компилируется в машинный код во время выполнения программы.
   * **Garbage Collector (GC):** CLR включает сборщик мусора, который автоматически управляет памятью, освобождая ресурсы, необходимые для объектов, которые больше не используются.
   * **Управление исключениями и обработка ошибок:** CLR предоставляет механизм для обработки исключений, что обеспечивает более безопасное выполнение программ.
2. **Библиотеки классов (Class Libraries):** Библиотеки классов .NET предоставляют набор готовых классов и методов, которые упрощают разработку. Они включают в себя функциональность для работы с файлами, сетями, базами данных и многими другими аспектами приложений.
3. **ASP.NET:** ASP.NET представляет собой фреймворк для создания веб-приложений на платформе .NET. Он предоставляет инфраструктуру для обработки HTTP-запросов, управления состоянием приложения и многие другие возможности для веб-разработки.

Общий поток работы включает разработку приложения на языке программирования, таком как C#, использование библиотек классов .NET для решения задач, а затем выполнение приложения в среде CLR, которая обеспечивает управление памятью, компиляцию кода в машинный код и другие аспекты выполнения программы.

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

**Структура управляемого модуля - Portable Executable (PE):**

Portable Executable (PE) - это формат файла, используемый для исполняемых и библиотечных файлов в операционных системах Windows, включая те, которые содержат управляемый код для .NET Framework. Вот основные компоненты структуры управляемого модуля PE:

1. **DOS Header:**
   * Содержит информацию, совместимую с форматом DOS (Disk Operating System), что обеспечивает обратную совместимость.
2. **PE Signature:**
   * Подпись, которая указывает на начало PE-заголовка.
3. **COFF (Common Object File Format) Header:**
   * Содержит информацию о секциях, символах и других атрибутах объектного файла.
4. **Optional Header:**
   * Включает в себя информацию о версии операционной системы, размере изображения, точке входа и другие параметры.
5. **Data Directories:**
   * Содержит адреса таблиц, таких как таблица импорта и таблица экспорта, которые указывают на различные части модуля.
6. **Section Headers:**
   * Описывают каждую секцию файла, такую как код, данные, ресурсы и другие.

**Исполнение сборки:**

Исполнение сборки в .NET происходит следующим образом:

1. **Компиляция:** Исходный код на управляемом языке (например, C#) компилируется в промежуточный язык Common Intermediate Language (CIL) с использованием компилятора языка .NET.
2. **Создание сборки:** Компилятор создает сборку, которая включает в себя метаданные (информацию о типах, методах, свойствах и т.д.) и исполняемый код в формате CIL.
3. **Метаданные и CIL в сборке:** Сборка содержит метаданные, описывающие типы и их взаимосвязи, а также исполняемый код в формате CIL. Метаданные обеспечивают управляемую среду выполнения информацией о структуре программы.
4. **JIT-компиляция:** Во время выполнения сборка передается в среду выполнения CLR, которая включает JIT-компилятор. JIT-компилятор преобразует CIL в машинный код, специфичный для конкретной архитектуры процессора, что позволяет выполнять программу.
5. **Выполнение:** Полученный машинный код выполняется на целевой системе. В процессе выполнения CLR обеспечивает управление памятью, обработку исключений, сборку мусора и другие аспекты управляемого выполнения.

**CIL (Common Intermediate Language):**

CIL - это промежуточный язык, который используется в процессе компиляции кода .NET. Важные особенности CIL включают:

* **Независимость от платформы:** CIL предоставляет промежуточный уровень абстракции, что делает код переносимым между различными архитектурами процессоров.
* **Объектная модель:** CIL поддерживает объектную модель, что обеспечивает поддержку объектно-ориентированного программирования.
* **Строгая типизация:** CIL является строго типизированным языком, что повышает безопасность и производительность.

Использование CIL и управляемой среды выполнения CLR делает .NET платформой, обеспечивающей высокий уровень абстракции, безопасность и переносимость кода.

Начало формы

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

**Common Type System (CTS):**

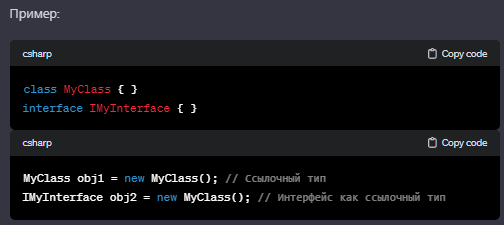
Common Type System (CTS) - это стандарт, определенный в рамках .NET Framework, который обеспечивает единый набор правил для представления типов данных в различных языках программирования, использующих платформу .NET. CTS обеспечивает совместимость типов между языками, что позволяет объектам, созданным на одном языке, взаимодействовать с объектами, созданными на других языках.

**Типы данных C# и их классификация:**

В C# типы данных делятся на две основные категории: ссылочные типы и типы значений.

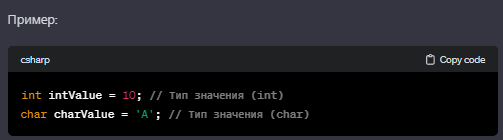
**1. Ссылочные типы (Reference Types):**

* **Классы (Classes):** Определенные пользователем типы данных.
* **Интерфейсы (Interfaces):** Определяют набор методов, которые класс должен реализовать.
* **Массивы (Arrays):** Коллекции элементов одного типа.
* **Делегаты (Delegates):** Типы, представляющие методы, которые можно присваивать и передавать как параметры.



**2. Типы значений (Value Types):**

* **Примитивные типы данных:** int, float, double, char, bool и другие.
* **Структуры (Structures):** Определенные пользователем типы данных с переменными-членами.
* **Перечисления (Enumerations):** Используются для определения именованных констант.

**Важно отметить:**

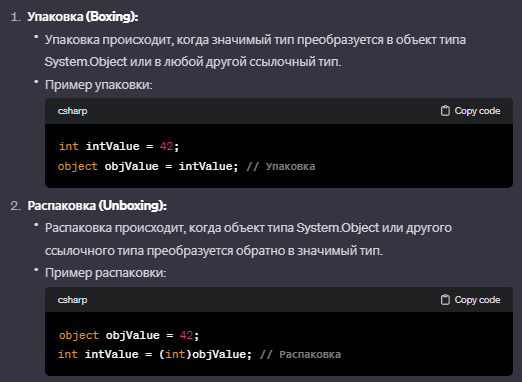
* Ссылочные типы хранят ссылку на объект, в то время как типы значений хранят само значение.
* Ссылочные типы и типы значений наследуют от System.Object, что делает их совместимыми в CTS.

Оба типа, ссылочные и типы значений, подчиняются правилам CTS, что обеспечивает их взаимодействие и совместимость в различных контекстах .NET Framework.

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

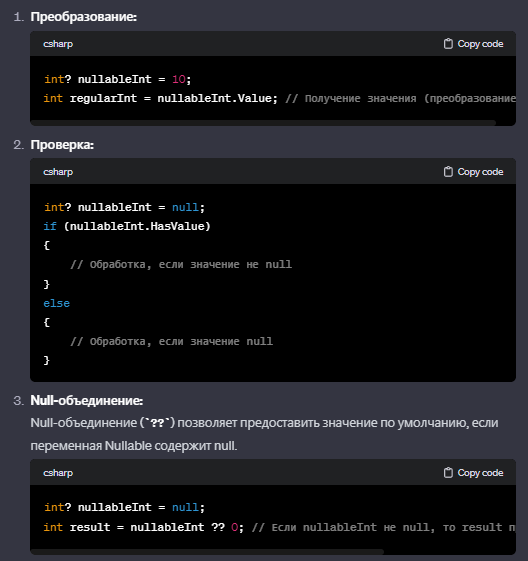
**Упаковка и распаковка типов:**

Упаковка (boxing) и распаковка (unboxing) - это процессы преобразования между значимыми типами (типами значений) и ссылочными типами в C#.



**Типы Nullable:**

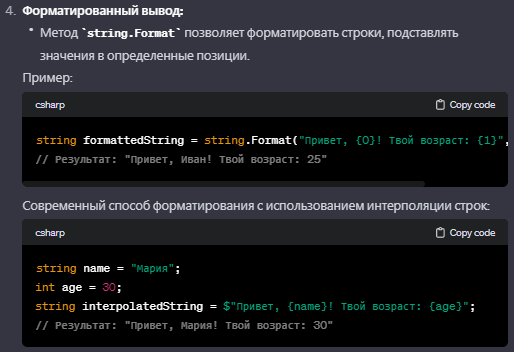
Типы Nullable предоставляют возможность представления значений, которые могут быть null, для значимых типов данных. Примером является **int?** (Nullable<int>), где **int?** может содержать значение типа **int** или **null**.

Типы Nullable полезны, например, при работе с базами данных, где столбцы могут содержать null значения, или при обработке данных, где некоторые значения отсутствуют. Они позволяют четко различать между отсутствующим значением и значением по умолчанию.

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

**Тип данных string в C#:**

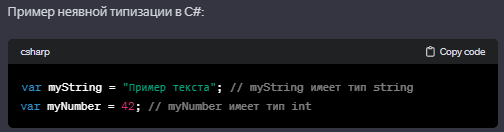
**string** - это тип данных, предназначенный для представления текстовых строк. Вот некоторые особенности и операции, связанные с типом **string**:



Тип данных **string** в C# предоставляет множество возможностей для работы с текстовой информацией, и разнообразные методы обеспечивают удобство работы с операциями над строками.

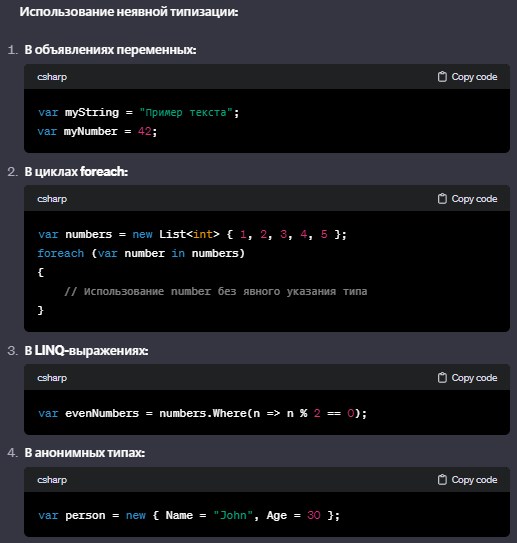
6. Неявная типизация – назначение и использование.

Неявная типизация, также известная как var (в C#), представляет собой механизм, который позволяет компилятору самостоятельно определить тип переменной на основе выражения, с которым она инициализируется. Вместо явного указания типа переменной, вы можете использовать ключевое слово var.



**Назначение неявной типизации:**

1. **Краткость кода:**
   * Использование неявной типизации уменьшает количество текста, который нужно писать, делая код более компактным.
2. **Увеличение читаемости:**
   * Использование var может сделать код более читаемым, особенно если тип переменной легко понять из контекста инициализации.
3. **Уменьшение повторений:**
   * При использовании LINQ-запросов или других сценариев, где типы могут быть сложными, неявная типизация может сэкономить от повторения типов в объявлениях переменных.



**Важные замечания:**

* Тип переменной, объявленной с использованием неявной типизации, определяется компилятором в момент компиляции, и после этого тип становится фиксированным.
* Неявная типизация не подходит в случаях, когда тип переменной неясен или когда четкость и явное указание типа предпочтительны.

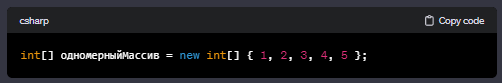
Используйте неявную типизацию там, где она облегчает читаемость кода, снижает его избыточность и подходит для конкретного контекста.

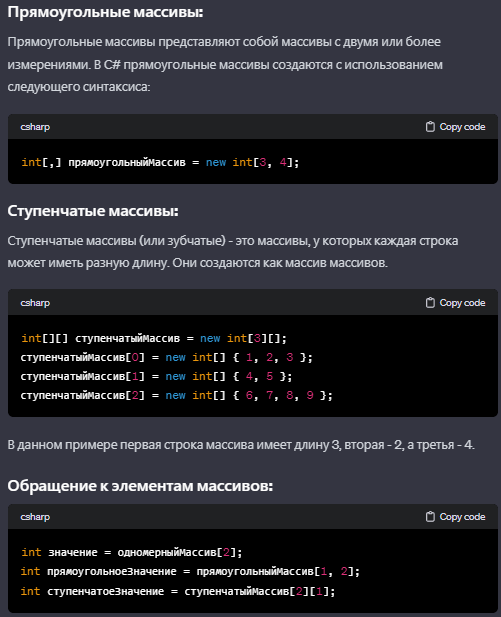
7. Массивы C# одномерные, прямоугольные и ступенчатые.

В C# существует несколько типов массивов, включая одномерные, прямоугольные и ступенчатые массивы.

**Одномерные массивы:**

Одномерные массивы являются наиболее простым типом и представляют собой набор элементов одного типа данных, хранящихся под одним и тем же именем.





**Важные замечания:**

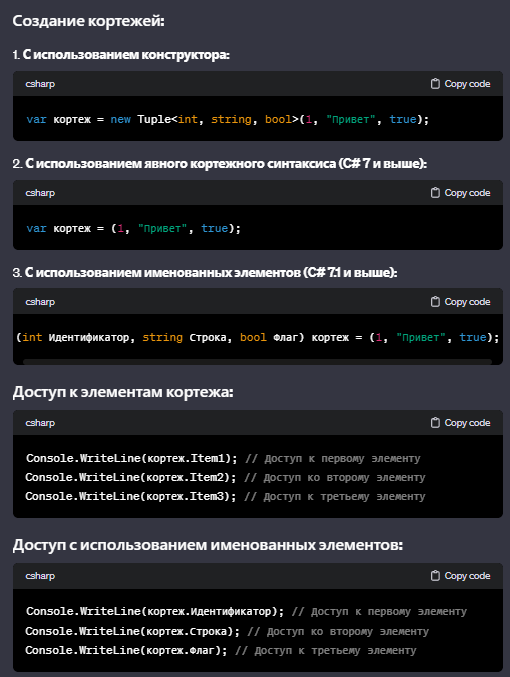
* Индексация в C# начинается с нуля.
* Одномерные массивы - это просто упорядоченные наборы данных.
* Прямоугольные массивы имеют два индекса (строки и столбцы).
* Ступенчатые массивы предоставляют гибкость в отношении длины каждой строки.

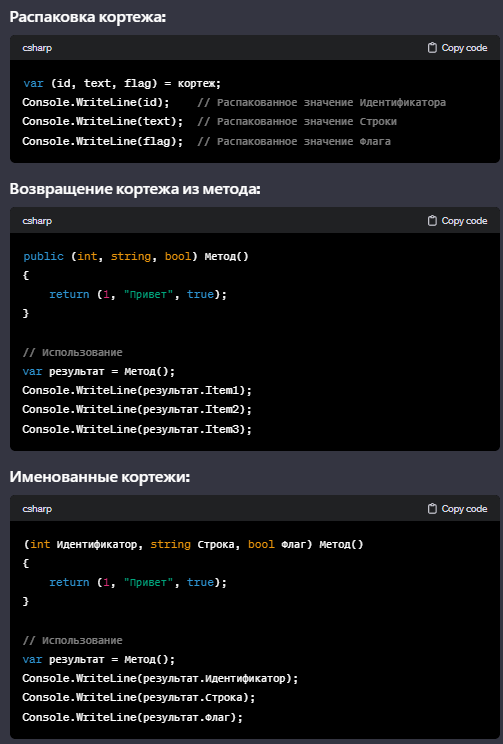
Выбор между различными типами массивов зависит от требований вашего приложения. Одномерные массивы хороши для простых наборов данных, в то время как прямоугольные и ступенчатые массивы предоставляют дополнительные возможности для структурирования сложных данных.

8. Понятие кортежей. Свойства, создание

**Кортежи в C#:**

Кортежи представляют собой структурированный набор значений, который может быть использован для возврата нескольких значений из метода, а также для временного хранения данных. Введение кортежей упрощает работу с данными, которые естественно группируются вместе.





Кортежи в C# предоставляют удобный способ работы с несколькими значениями, не требуя явного создания отдельных структур или классов. Их использование особенно удобно, когда возвращается несколько значений из метода или необходимо временно сгруппировать данные.

9. Принципы объектно-ориентированного программирования.

Объектно-ориентированное программирование (ООП) - это парадигма программирования, основанная на концепции объектов, которые объединяют данные и методы (функции), работающие с этими данными, в единый элемент. Принципы ООП обеспечивают организацию кода, облегчают его понимание и поддержку. Вот основные принципы ООП:

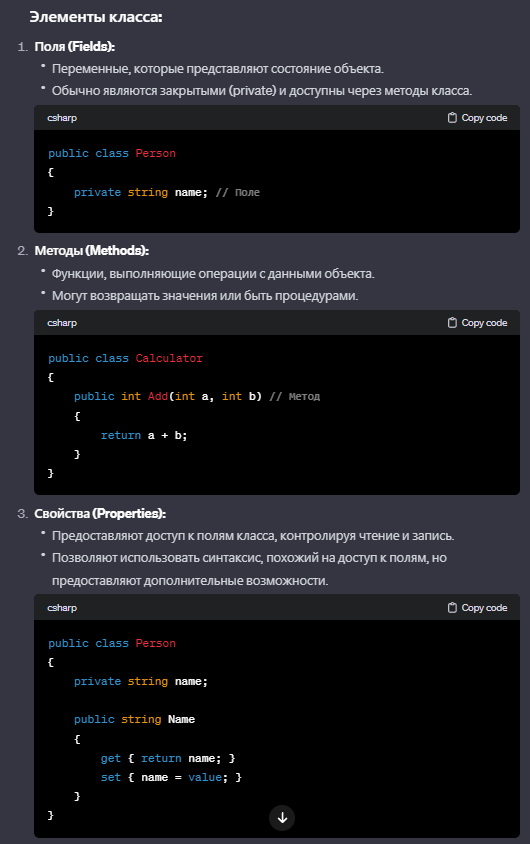
1. **Инкапсуляция (Encapsulation):**
   * Идея заключается в том, чтобы объединить данные и методы, работающие с этими данными, внутри объекта.
   * Объект скрывает детали реализации от внешнего мира и предоставляет интерфейс для взаимодействия с этими данными и методами.
2. **Наследование (Inheritance):**
   * Одна из ключевых идей ООП, которая позволяет создавать новый класс на основе существующего (родительского) класса.
   * Новый класс, называемый подклассом или производным классом, наследует свойства и методы родительского класса.
   * Позволяет повторно использовать код и создавать иерархии классов.
3. **Полиморфизм (Polymorphism):**
   * Возможность объектов разных типов обрабатываться как объекты одного и того же базового типа.
   * Полиморфизм может быть реализован через перегрузку методов, виртуальные методы и интерфейсы.
   * Упрощает и унифицирует вызов методов и обработку данных.
4. **Абстракция (Abstraction):**
   * Абстракция позволяет создавать упрощенные модели реального мира, выделяя основные характеристики и игнорируя ненужные детали.
   * Позволяет сконцентрироваться на ключевых аспектах объекта и игнорировать детали его реализации.
5. **Принцип единственной ответственности (Single Responsibility Principle - SRP):**
   * Каждый класс должен иметь одну и только одну причину для изменения.
   * Класс должен быть ответственным только за одну важную функциональность.
6. **Принцип открытости/закрытости (Open/Closed Principle - OCP):**
   * Программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для модификации.
   * Модификации в существующем коде должны быть минимальными, а новая функциональность должна добавляться путем расширения.
7. **Принцип подстановки Барбары Лисков (Liskov Substitution Principle - LSP):**
   * Объекты базового класса могут быть заменены объектами производного класса без изменения корректности программы.
   * Если S - это подтип T, то объекты типа T могут быть заменены объектами типа S без изменения свойств программы.
8. **Принцип инверсии зависимостей (Dependency Inversion Principle - DIP):**
   * Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций.
   * Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

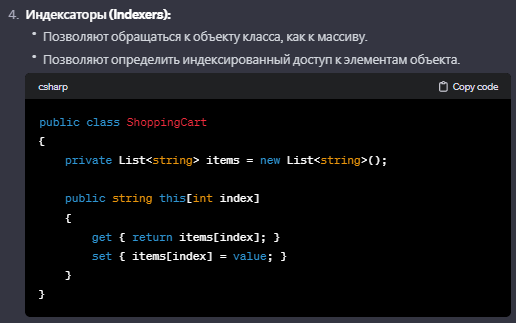
Эти принципы ООП предоставляют базовые стратегии для организации кода, сделав его более гибким, удобным для понимания, расширяемым и поддерживаемым.

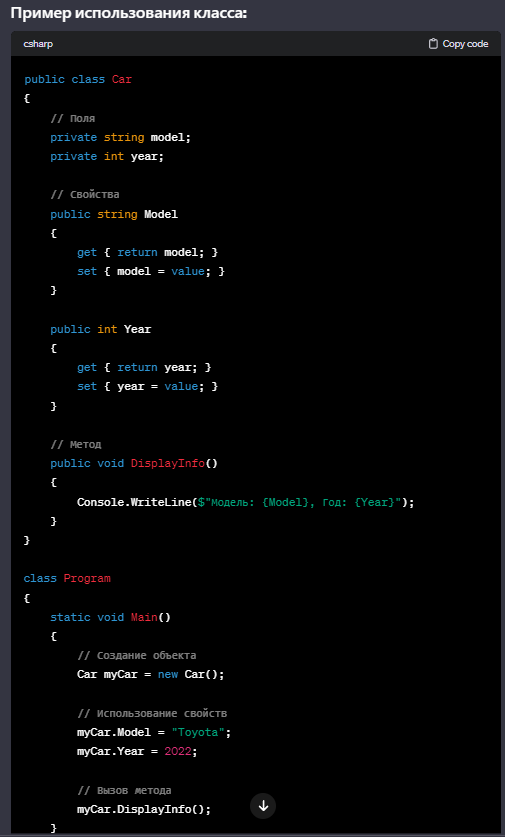
10. Класс. Элементы класса. Свойства и индексаторы.

**Класс в C#:**

Класс - это шаблон для создания объектов. Он определяет данные и методы, которые работают с этими данными. Класс является основным строительным блоком объектно-ориентированного программирования.





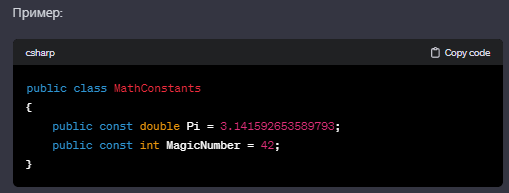


В этом примере класс **Car** содержит поля **model** и **year**, свойства **Model** и **Year**, метод **DisplayInfo**, а также демонстрирует создание и использование объекта этого класса. Классы являются основными строительными блоками для организации кода в объектно-ориентированном программировании.

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

**Константы в классе:**

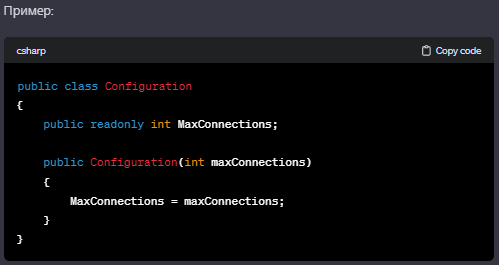
Константы в C# представляют собой значения, которые не могут быть изменены после инициализации. В классе они объявляются с использованием ключевого слова **const**. Константы должны быть проинициализированы во время компиляции.



В этом примере **Pi** и **MagicNumber** - это константы, и их значения не могут быть изменены в процессе выполнения программы.

**Поля только для чтения:**

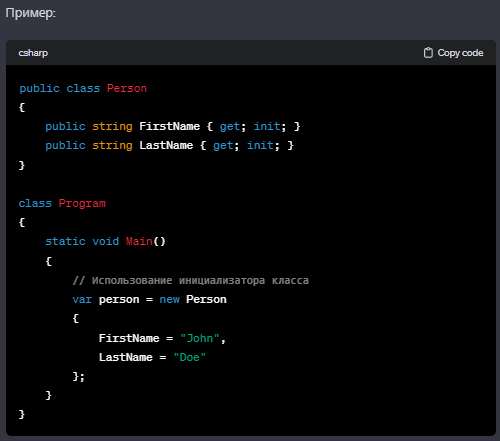
Поля только для чтения в C# могут быть инициализированы только в момент объявления или в конструкторе класса. Они не могут быть изменены после инициализации.



В этом примере **MaxConnections** - это поле только для чтения, которое может быть установлено только в конструкторе.

**Инициализаторы класса (Class Initializers):**

В C# 9.0 и новее появилась возможность использовать инициализаторы класса для удобной инициализации свойств и полей в момент создания объекта.



В этом примере **FirstName** и **LastName** инициализируются с использованием инициализатора класса при создании объекта **Person**. Оператор **init** указывает, что свойства могут быть установлены только во время инициализации.

Эти концепции - константы, поля только для чтения и инициализаторы класса - предоставляют различные способы создания и использования значений внутри классов с учетом их изменяемости и инициализации.

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

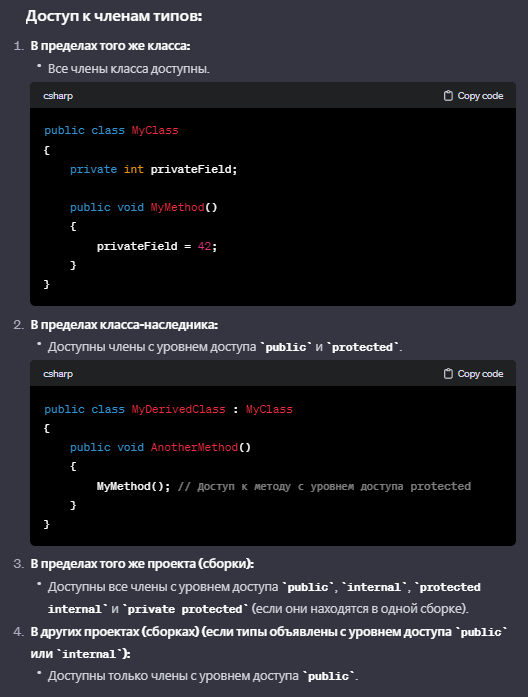
**Спецификаторы доступа в C#:**

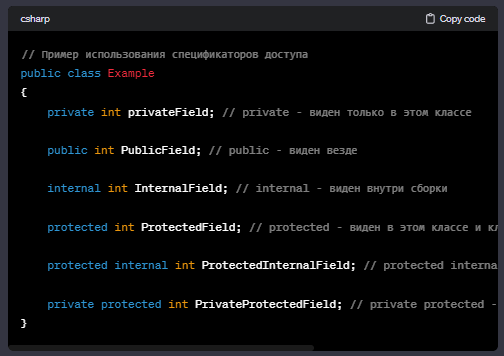
Спецификаторы доступа определяют уровень видимости классов, полей, методов и других членов типов в C#. В языке C# используются следующие спецификаторы доступа:

1. **public:**
   * Член виден везде, как внутри сборки, так и за ее пределами.
2. **private:**
   * Член виден только внутри класса, в котором он объявлен.
3. **protected:**
   * Член виден внутри своего класса и в классах-наследниках.
4. **internal:**
   * Член виден в пределах текущей сборки.
5. **protected internal:**
   * Член виден в пределах текущей сборки и в классах-наследниках, даже если они находятся в других сборках.
6. **private protected (C# 7.2 и новее):**
   * Член виден только внутри своего класса и в классах-наследниках внутри текущей сборки.

**Видимость типов:**

* **public:**
  + Тип виден везде.
* **internal:**
  + Тип виден только в пределах текущей сборки.
* **public internal:**
  + Тип виден везде и в пределах текущей сборки.



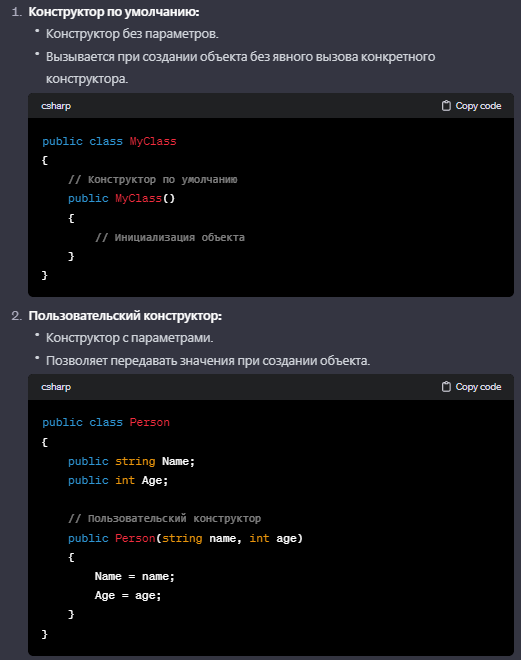


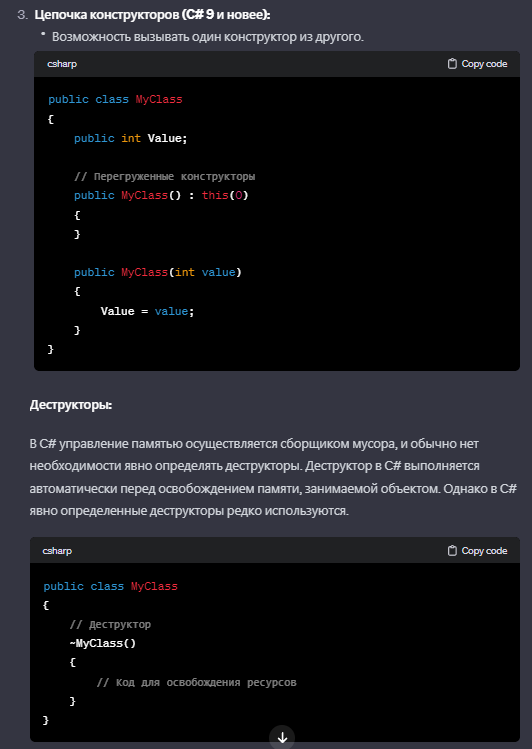
Используйте спецификаторы доступа для эффективного контроля видимости и доступа к членам типов в C#.

13. Класс. Конструкторы и их свойства. Деструкторы

**Конструкторы в C#:**

Конструктор - это метод класса, который выполняется при создании объекта. Он используется для установки начальных значений полей объекта и выполнения других необходимых действий при его создании. В C# можно определить несколько видов конструкторов:







Используйте деструкторы и методы **Dispose** осторожно, и предпочтительно использовать **using** или **try-finally** для управления ресурсами.

14. Класс и методы System.Object.

**System.Object** является базовым классом для всех типов в C#. Все типы в C# наследуются от **System.Object** непосредственно или косвенно. В **System.Object** определены несколько важных методов, которые могут быть переопределены в производных классах. Вот несколько из них:



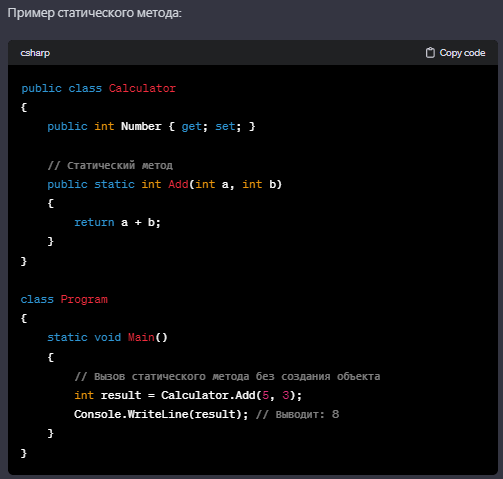


Эти методы предоставляют базовую функциональность для всех объектов в C# и могут быть переопределены в производных классах, чтобы предоставить пользовательскую логику.

15. Статические методы и статические конструкторы класса.

**Статические методы:**

В C# статические методы принадлежат к типу, а не к конкретному объекту этого типа. Они могут вызываться без создания экземпляра класса. Для определения статического метода используется ключевое слово **static**.

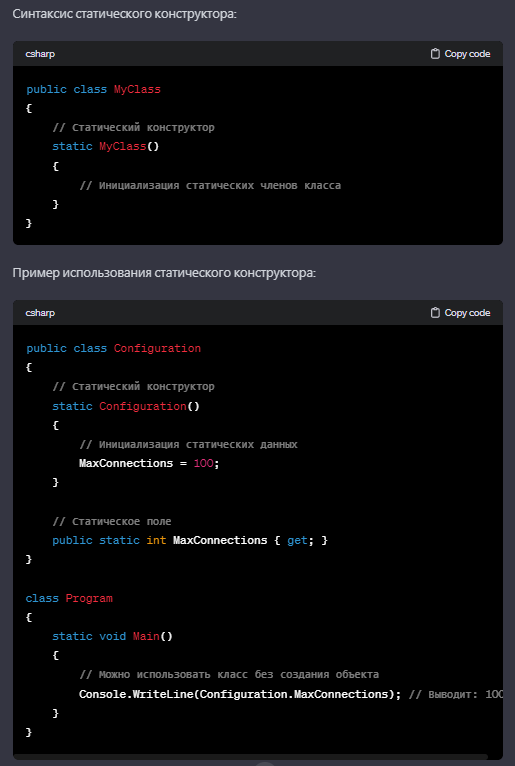


Статические методы могут использоваться для предоставления функциональности, которая не требует состояния объекта.

**Статические конструкторы:**

Статический конструктор вызывается при первом обращении к классу, а не при создании экземпляра. Он может использоваться для выполнения инициализации, которая требуется только один раз для всего класса.

Синтаксис статического конструктора:

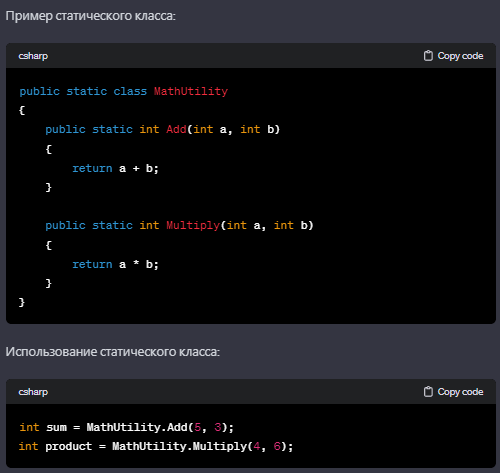


Статические конструкторы удобны для инициализации статических данных или выполнения других операций, которые должны быть выполнены единожды при первом обращении к классу. Важно отметить, что статический конструктор вызывается автоматически и не может быть вызван явно.

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

**Статические классы:**

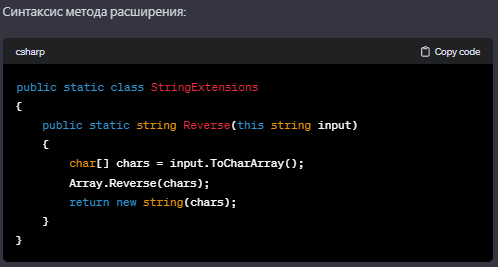
Статический класс в C# - это класс, который не может быть инстанциирован, и все его члены являются статическими. Он используется, чтобы предоставить функциональность, которая не требует создания экземпляра объекта.



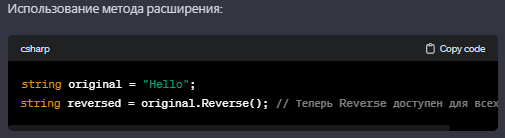
Статические классы удобны для группировки функциональности, которая не зависит от состояния объекта.

**Методы расширения:**

Методы расширения позволяют добавлять новые методы к существующим типам без изменения их кода. Они могут быть использованы только с сборками, в которых они определены, и должны быть размещены в статическом, необобщенном статическом классе.



Обратите внимание на ключевое слово **this** перед первым параметром. Это указывает, что этот метод является методом расширения для типа, указанного перед **this**. В данном примере, метод **Reverse** становится доступным для всех строк в вашем коде.



**Правила определения методов расширения:**

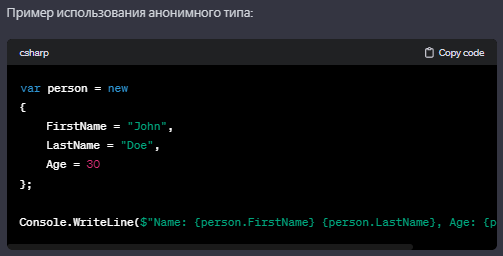
1. Метод расширения должен быть определен в статическом, необобщенном статическом классе.
2. Метод расширения должен быть статическим.
3. Первый параметр метода расширения указывает тип, который расширяется, и должен быть предварен ключевым словом **this**.
4. Метод расширения не может изменять тип, который он расширяет.
5. Методы расширения могут быть определены только в пространствах имен или модулях, а также в том же проекте, где они используются.

17. Анонимные типы.

Анонимные типы - это типы данных в C#, которые предоставляют простой способ создания объектов с набором свойств в процессе выполнения кода. Они особенно удобны, когда вам нужно создать временные объекты без явного определения класса.

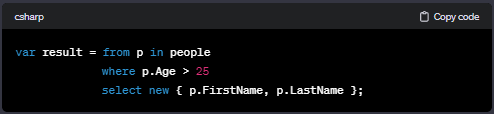
Основные характеристики анонимных типов:

1. **Создание в процессе выполнения:**
   * Анонимные типы создаются без явного определения класса в процессе выполнения кода.
2. **Определение свойств:**
   * Анонимные типы предоставляют свойства, которые определяются непосредственно при создании объекта.
3. **Неявная типизация:**
   * Типы свойств определяются неявно компилятором на основе предоставленных значений.



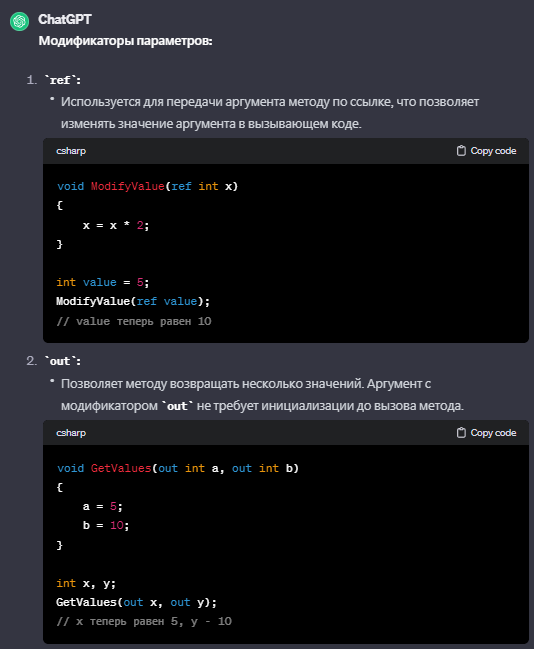
В этом примере создается анонимный тип с тремя свойствами: **FirstName**, **LastName** и **Age**. Эти свойства определяются неявно при создании объекта анонимного типа.

Анонимные типы часто используются, например, при выполнении запросов LINQ, где результат запроса не является типизированным и необходимо быстро создать временные объекты с определенными свойствами.

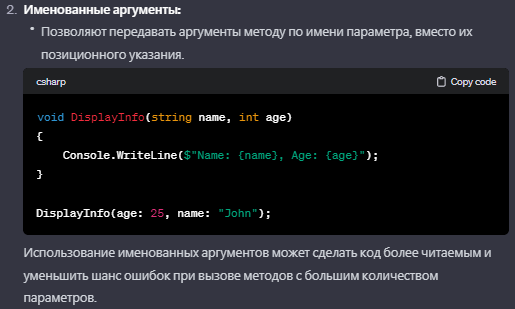


В данном примере создаются анонимные объекты для каждого элемента коллекции **people**, содержащие только свойства **FirstName** и **LastName**.

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



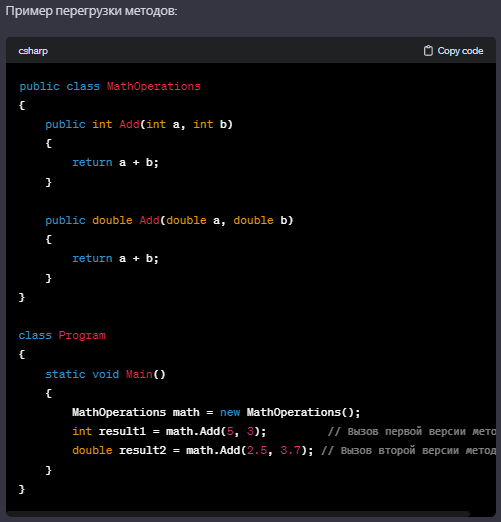




19. Перегрузка методов и операторов. Правила перегрузки операторов.

**Перегрузка методов:**

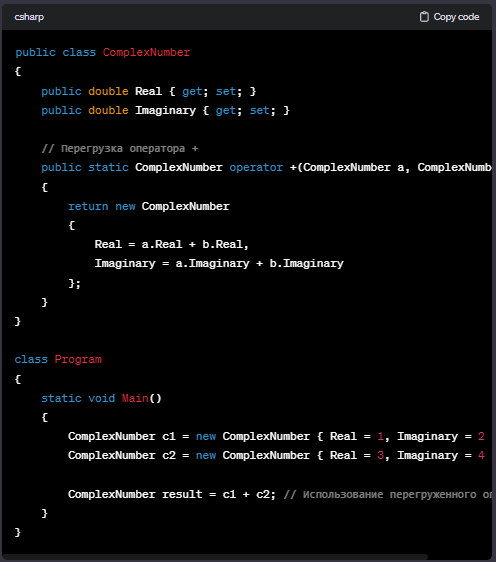
Перегрузка методов в C# позволяет определить несколько методов с одним и тем же именем в одном классе, но с разными параметрами. При вызове метода компилятор выбирает наиболее подходящий по сигнатуре метод.



**Перегрузка операторов:**

В C# можно перегружать операторы для пользовательских типов данных. Это позволяет определить, каким образом должны выполняться операции над объектами данного типа.

Пример перегрузки оператора **+** для пользовательского типа:



**Правила перегрузки операторов:**

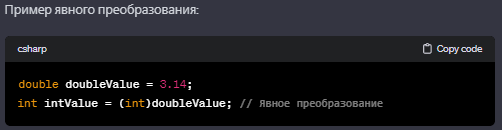
1. **Операторы, которые можно перегружать:**
   * **+**, **-**, **\***, **/**, **%**, **&**, **|**, **^**, **~**, **!**, **=**, **<**, **>**, **<=**, **>=**, **==**, **!=**, **<<**, **>>**.
2. **Все перегруженные операторы являются статическими методами.**
   * Например, перегрузка оператора **+** представлена как **public static ComplexNumber operator +(ComplexNumber a, ComplexNumber b)**.
3. **Нельзя создать новый оператор.**
   * Можно перегрузить существующий, но нельзя добавить новый оператор, не поддерживаемый языком.
4. **Перегрузка оператора == также должна включать перегрузку оператора !=.**
   * Аналогично, если перегружается **<**, то должен быть перегружен и **>**.
5. **Приоритет и ассоциативность операторов не изменяются при их перегрузке.**
   * Например, если оператор **+** является левоассоциативным, то и его перегрузка также будет левоассоциативной.
6. **Не все операторы могут быть перегружены в структурах.**
   * Например, операторы **+=**, **-=** и т. д. нельзя перегружать для структур.

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

В C# операции преобразования типа делятся на явные и неявные формы. Они позволяют преобразовывать значения между разными типами данных.

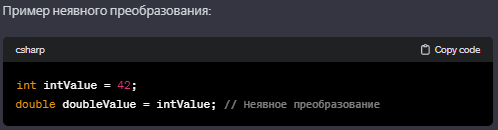
**Явные преобразования:**

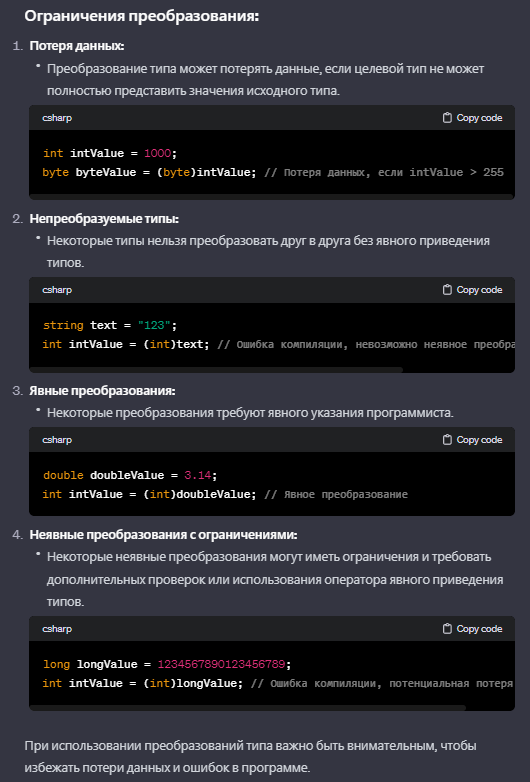
**Явное преобразование (explicit conversion)** требует явного указания программиста и может привести к потере данных.



**Неявные преобразования:**

**Неявное преобразование (implicit conversion)** выполняется автоматически компилятором и не требует явного указания программиста. Оно безопасно и не приводит к потере данных.

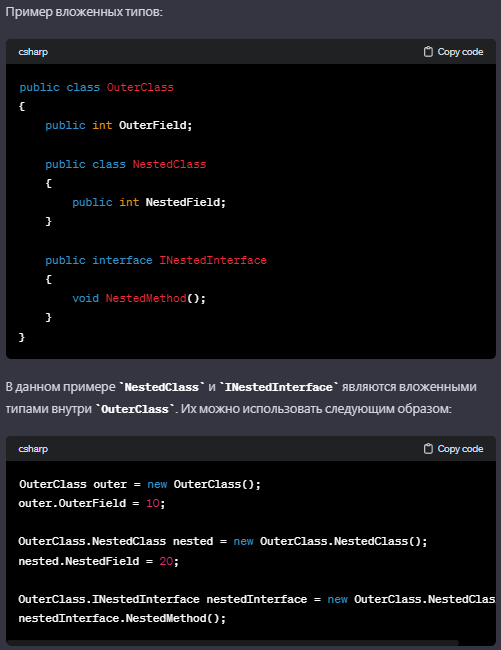




21. Вложенные типы. Вложенные объекты

**Вложенные типы:**

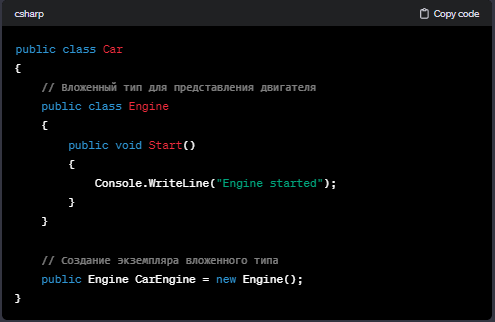
В C# вложенные типы (nested types) — это типы, которые объявлены внутри другого типа. Они могут быть классами, интерфейсами, структурами или перечислениями. Вложенные типы обычно используются для организации кода, повышения инкапсуляции и сокрытия реализации.



**Вложенные объекты:**

Когда говорят о вложенных объектах, обычно имеют в виду создание объектов внутри других объектов. Это может включать в себя создание экземпляров вложенных типов, композицию объектов и т. д.

Пример вложенного объекта с использованием вложенных типов:



В этом примере **Engine** является вложенным типом внутри класса **Car**, и мы создаем экземпляр **Engine** внутри объекта **Car**. Такой подход позволяет структурировать код и улучшить читаемость. Использование вложенных объектов может быть удобным, если некоторые компоненты объекта логически связаны и могут быть описаны в рамках вложенных типов.

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. Структур 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.