**1. Чем класс отличается от структуры?**

В C#, структура (structure) — это пользовательский тип данных, который представляет собой совокупность переменных различных типов, объединенных под одним именем. Структуры являются типами значений, и их экземпляры хранятся непосредственно в стеке памяти или вложенно в другие объекты, в отличие от объектов классов, которые хранятся в куче и передаются по ссылке.

|  |  |  |
| --- | --- | --- |
|  | Структура | Класс |
| Тип | Структуры являются типами значений. Это означает, что экземпляры структур хранятся непосредственно в памяти, а не по ссылке, как объекты классов. При присваивании значения структуры другой структуре происходит копирование данных, т.е. при присваивании копируются все её члены. | Классы являются типами ссылки. Экземпляры классов хранятся в куче, и переменные содержат ссылки на эти объекты. Когда вы присваиваете значение одного объекта класса другой переменной, копируется только ссылка на объект, а не сам объект. Оба объекта переменных будут указывать на один и тот же экземпляр класса в памяти. |
| Создание | Структуры могут быть созданы без вызова конструктора, и все их члены инициализируются значениями по умолчанию. | Для создания объекта класса обычно требуется использование конструктора. Класс может иметь явные конструкторы без параметров и с параметрами. |
| Наследование | Структуры не поддерживают наследование друг от друга, и они не могут быть базовыми классами для других типов. | Классы могут быть наследованы от других классов, что обеспечивает механизм наследования и полиморфизма. |
| Конструкторы | Структуры не могут иметь явных конструкторов без параметров. |  |
| Функционал | Структуры могут содержать поля, свойства, методы, конструкторы, но они не могут содержать деструкторы, события или индексаторы. | Классы могут содержать дополнительные элементы, такие как деструкторы, события, индексаторы, их члены могут быть объявлены с различными модификаторами доступа. |

Структуры обычно предпочтительны, когда объекты небольшие и легко копируются, а классы - когда необходима сложная логика, наследование или когда объекты большие и требуют эффективной работы с памятью через ссылки.

Пример структуры в C#:

 Пример класса в C#:



**2. Что может и чего не может быть в структуре?**

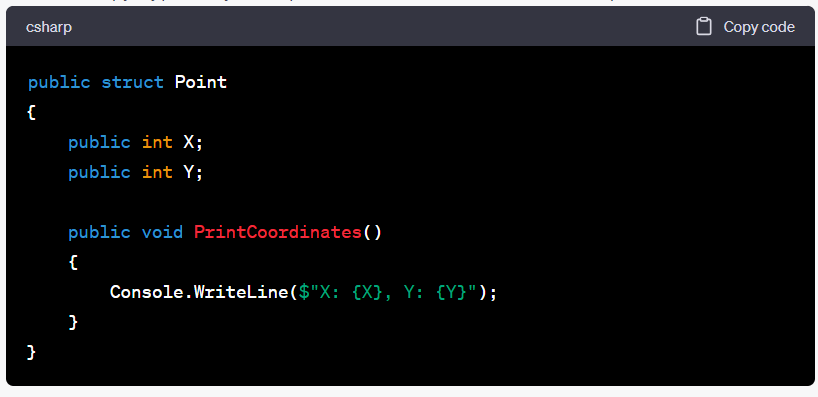
В C# структуры (structures) могут содержать различные элементы, но они также имеют некоторые ограничения и отличия от классов. Вот основные черты структур:

\*\*Что может быть в структуре:\*\*

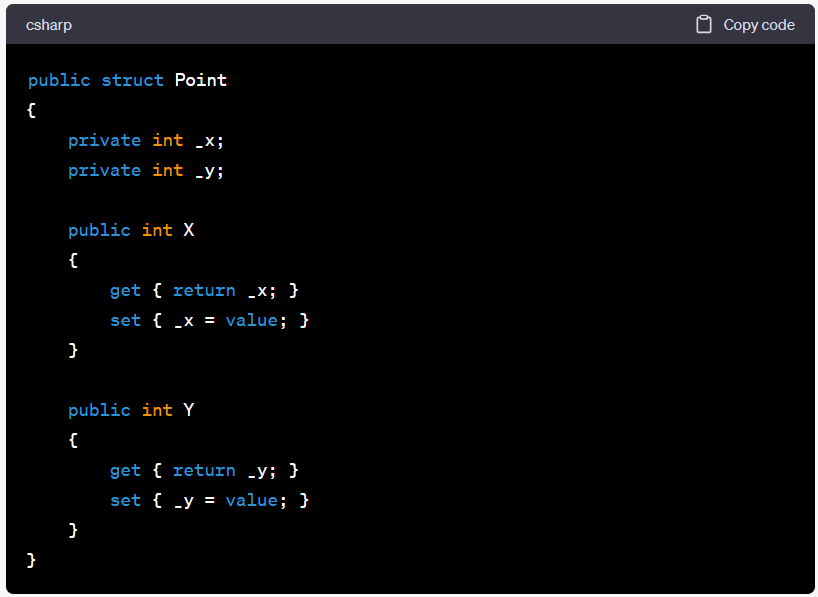
1. \*\*Поля данных (Fields):\*\* Структуры могут содержать поля для хранения данных различных типов.



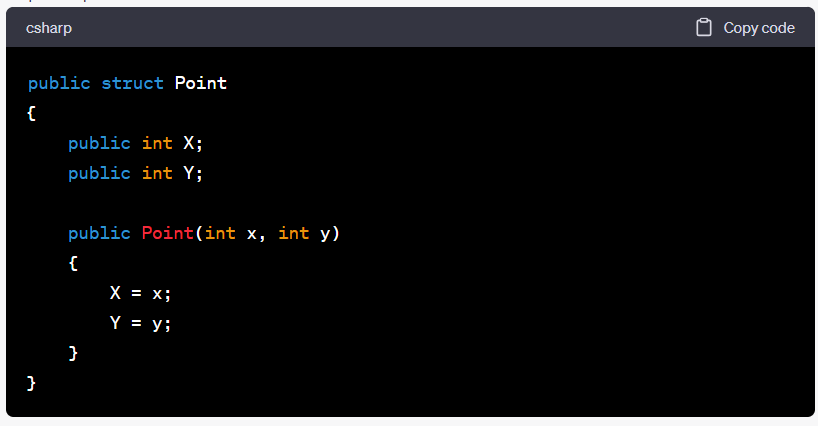
2. \*\*Методы:\*\* Структуры могут содержать методы для выполнения операций с данными.



3. \*\*Свойства (Properties):\*\* Структуры могут иметь свойства для доступа и изменения их данных.

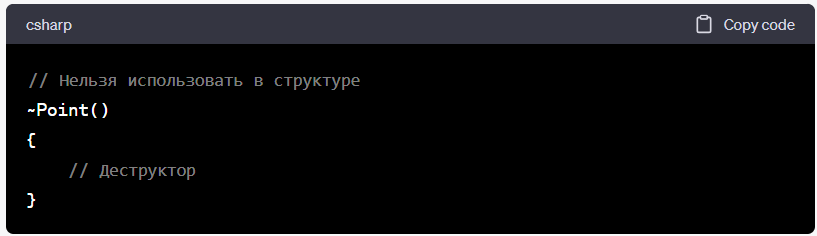


4. \*\*Конструкторы:\*\* Структуры могут иметь конструкторы, в том числе конструкторы с параметрами.

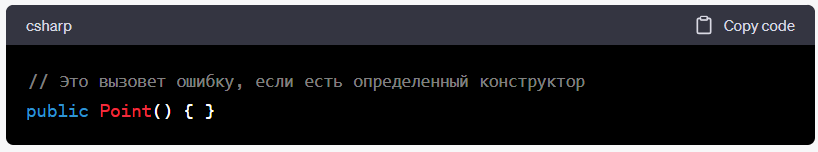


\*\*Чего не может быть в структуре:\*\*

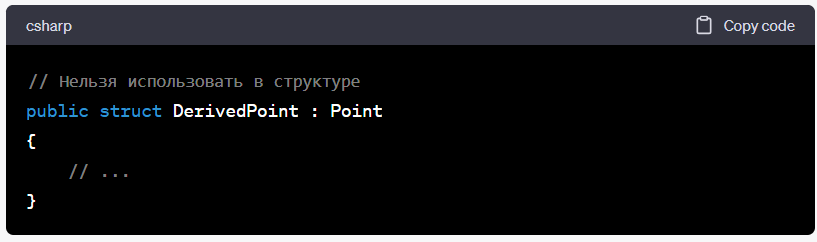
1. \*\*Деструкторы (Finalizers):\*\* Структуры не могут содержать деструкторы. Деструкторы применимы только к классам.



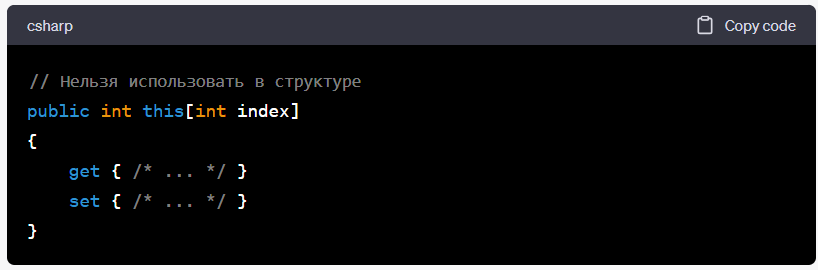
2. \*\*Явные конструкторы без параметров:\*\* Если структура имеет определенные конструкторы, то она не может содержать явных конструкторов без параметров.



3. \*\*Базовые структуры (Struct Base):\*\* Структуры не могут наследовать от других структур или классов.



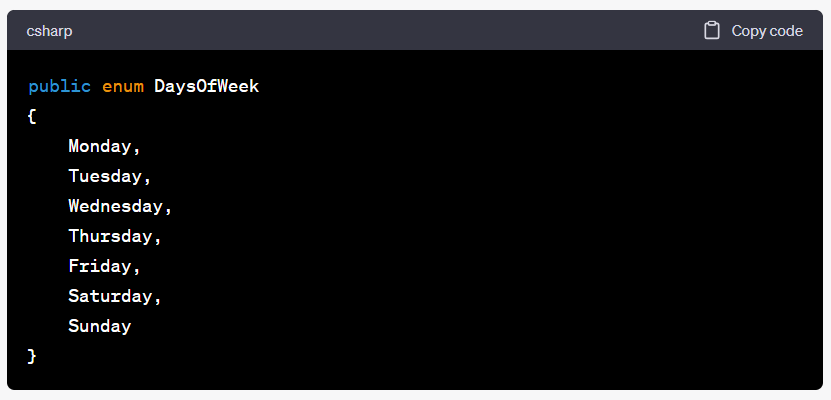
4. \*\*Индексаторы (Indexers):\*\* Структуры не могут содержать индексаторы.



**3. Что такое перечисление? Приведите пример определения и использования перечисления**

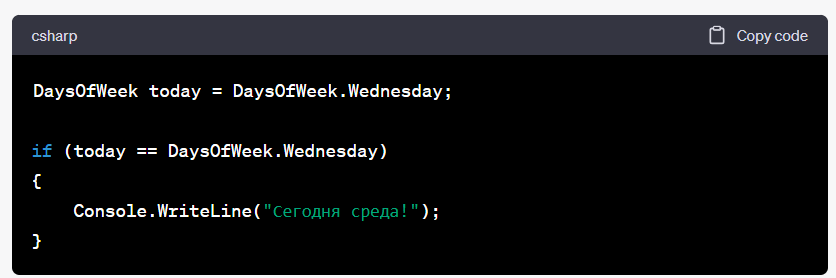
Перечисление (Enum) в C# представляет собой именованный набор константных значений, которые представляют собой имена для определенных целочисленных значений. В перечислении каждому имени сопоставляется целочисленное значение, которое по умолчанию увеличивается на 1 для каждого последующего элемента, но может быть изменено явным образом.

Пример простого перечисления:

****

В этом примере **DaysOfWeek** - это перечисление, которое представляет дни недели. Каждый элемент перечисления (**Monday**, **Tuesday**, и так далее) имеет целочисленное значение, причем первый элемент по умолчанию равен 0, второй - 1 и так далее.

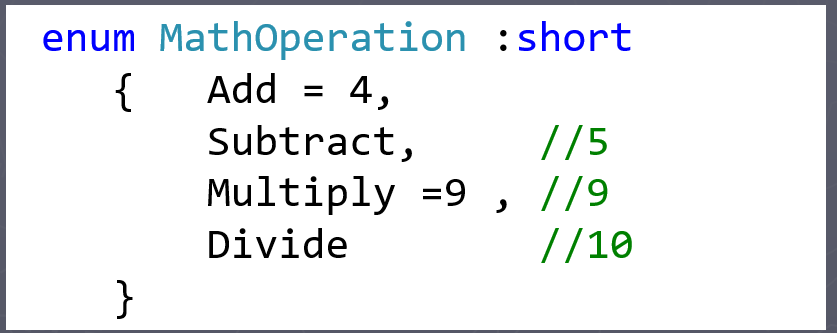
Пример использования перечисления:

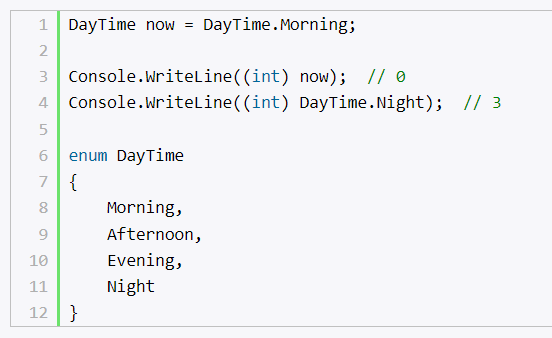


Создается переменная today типа DaysOfWeek и присваивается значение DaysOfWeek.Wednesday. Теперь today представляет текущий день как среду.

С использованием условного оператора if проверяется, является ли текущий день средой. Если условие выполняется (то есть today равно DaysOfWeek.Wednesday), то выводится сообщение "Сегодня среда!".

1. тип перечисления - целочисленный тип (byte, int, short, long)
2. По умолчанию используется тип int
3. каждому элементу перечисления присваивается целочисленное значение, 0, 1 т.д.
4. Можно определять явным образом





**4. Перечислите и поясните стандартные интерфейсы .Net?**

|  |  |  |  |
| --- | --- | --- | --- |
| Интерфейс | методы | Назначение | прим |
| ICloneable | object Clone() | Клонирование объектов (поверхностное или глубокое) |  |
| lEnumerable | (IEnumerator) GetEnumerator(); | перебор элементов необобщенной коллекции | основна для большинства коллекций |
| IEnumerator | Current  bool MoveNext()  void Reset() | перебор по необобщенной коллекции | можем перебирать объекты в цикле foreach |
| IComparable | int CompareTo( object obj ) | Сравнение объектов  Для выяснения порядка |  |
| IComparer | int Compare(object o1,  object o2); | Сравнение объектов | будут иметь больший приоритет |

1. \*\*ICloneable (Клонирование):\*\*

- \*\*Описание:\*\* Интерфейс `ICloneable` предоставляет метод `Clone()`, который позволяет создавать глубокие копии объектов. Глубокое копирование означает, что копируются и значения всех полей объекта, включая вложенные объекты, а не только ссылки на них.

- \*\*Использование:\*\* Реализация этого интерфейса позволяет объектам быть клонируемыми. Это удобно, когда требуется создать копию объекта без изменения исходного.

2. \*\*IEnumerable (Перечисление):\*\*

- \*\*Описание:\*\* Интерфейс `IEnumerable` представляет коллекцию объектов, которую можно перечислить. Он определяет метод `GetEnumerator()`, возвращающий объект, реализующий интерфейс `IEnumerator`.

- \*\*Использование:\*\* Реализация этого интерфейса позволяет объекту быть перечисляемым, что в свою очередь делает его доступным для использования в циклах `foreach` и других конструкциях, работающих с перечислениями.

3. \*\*IEnumerator (Итератор):\*\*

- \*\*Описание:\*\* Интерфейс `IEnumerator` предоставляет методы для итерации по элементам коллекции. Он содержит методы `MoveNext()`, `Reset()` и свойство `Current`, позволяющие последовательно перебирать элементы.

- \*\*Использование:\*\* Реализация этого интерфейса обеспечивает механизм для создания объекта, который умеет перебирать элементы коллекции. Этот объект возвращается методом `GetEnumerator()` интерфейса `IEnumerable`.

4. \*\*IComparable (Сравнение):\*\*

- \*\*Описание:\*\* Интерфейс `IComparable` определяет метод `CompareTo()`, который позволяет сравнивать текущий объект с другим. Возвращает отрицательное число, если текущий объект меньше, положительное — если больше, и ноль, если они равны.

- \*\*Использование:\*\* Реализация этого интерфейса позволяет объекту быть сравниваемым, что полезно, например, при сортировке элементов коллекции.

5. \*\*IComparer (Компаратор):\*\*

- \*\*Описание:\*\* Интерфейс `IComparer` предоставляет метод `Compare()`, который позволяет сравнивать два объекта. Это отличается от `IComparable`, который определяет сравнение для одного объекта.

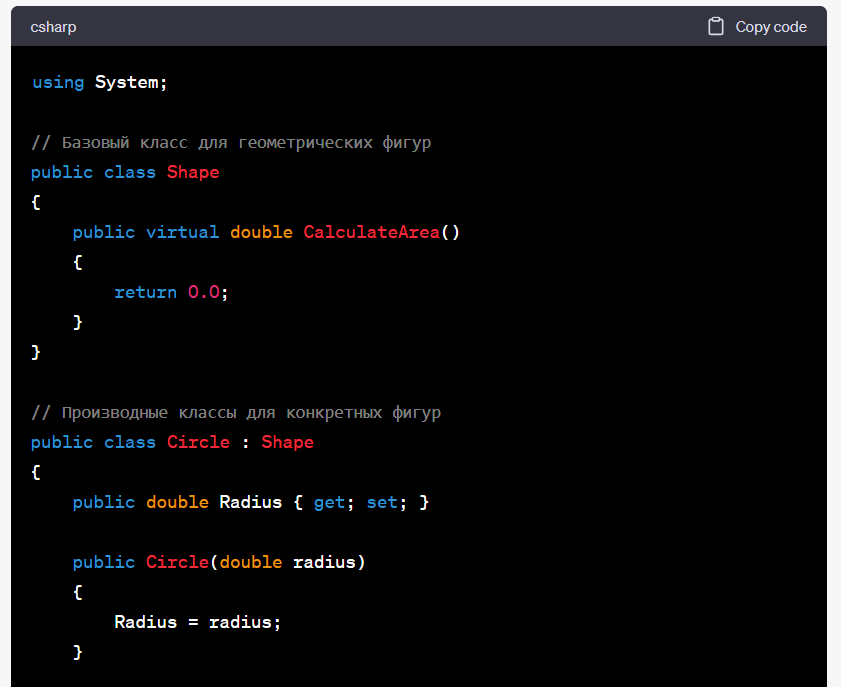
- \*\*Использование:\*\* Реализация этого интерфейса позволяет создавать отдельные классы, которые могут использоваться для сравнения объектов без изменения самих классов этих объектов.

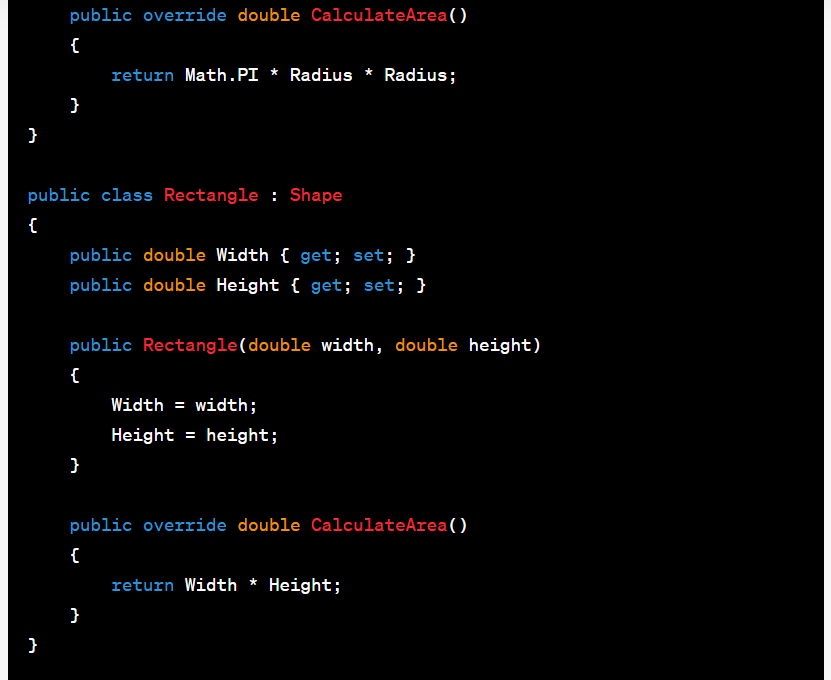
**7. Что такое полиморфизм? Перечислите его формы. Приведите примеры.**

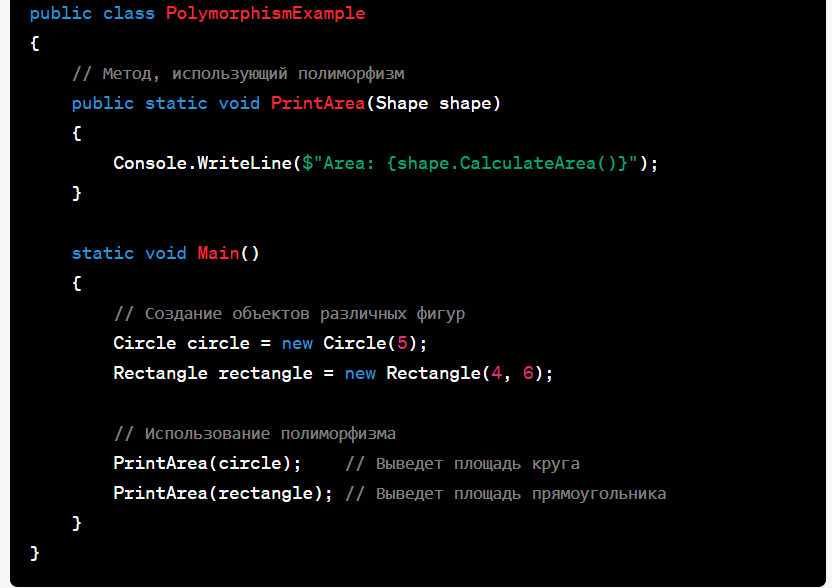
\*\*Полиморфизм\*\* - это свойство объектов в программировании, которое позволяет объектам разных классов предоставлять общий интерфейс, но реализовывать их по-разному. Проще говоря, это способ использовать один и тот же метод или свойство для объектов разных типов.

Например, у разных животных может быть одинаковый метод "издать звук", но каждый вид животных будет издавать свой уникальный звук.

* -поволяет методам классов иметь не одну, а несколько форм, и он необходим, когда у нас есть много классов, связанных друг с другом путем наследования
* - способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта
* Поддержка полиморфизма осуществляется через *виртуальные* функции, механизм перегрузки функций и операторов, а также обобщения







Статический и динамический полиморфизм - это две различные формы полиморфизма в объектно-ориентированных языках программирования. Вот основные различия между ними:

2. \*\*Связывание:\*\*

- \*\*Статический полиморфизм:\*\* Также известный как раннее (или статическое) связывание. Решение о том, какая версия метода будет вызвана, принимается на этапе компиляции.

- \*\*Динамический полиморфизм:\*\* Также известный как позднее (или динамическое) связывание. Решение о том, какой метод вызвать, принимается во время выполнения, что позволяет более гибкую замену объектов и использование интерфейсов.

4. \*\*Примеры:\*\*

- \*\*Статический полиморфизм:\*\* Примерами являются перегрузка методов (overloading) и шаблоны (в C++), где компилятор на основе сигнатур и контекста решает, какой метод использовать.

- \*\*Динамический полиморфизм:\*\* Примерами являются виртуальные методы и интерфейсы, где вызываемый метод определяется во время выполнения в зависимости от типа объекта.

Необходимые условия для реализации позднего связывания:

* классы должны образовывать иерархию наследования;
* в классах должны быть методы с одинаковой сигнатурой. Элементы (методы) производных классов должны перекрывать (override) соответствующие элементы (методы) базовых классов;
* элементы (методы) класса должны быть виртуальными, то есть должны быть обозначены ключевыми словами virtual, override.

**8. Зачем в классе определяют виртуальные методы?**

1. \*\*Полиморфизм:\*\*

- Главная цель виртуальных функций — обеспечить динамический полиморфизм.

- Виртуальные функции позволяют объектам различных классов использовать один и тот же интерфейс (название и сигнатура метода), что делает возможным вызывать методы, принадлежащие производным классам, через указатель (или ссылку) на базовый класс.

2. \*\*Переопределение методов:\*\*

- Виртуальные функции могут быть переопределены в производных классах.

- Это позволяет классам-наследникам предоставлять свою собственную реализацию методов, унаследованных от базового класса.

3. \*\*Динамическая диспетчеризация:\*\*

- Виртуальные функции обеспечивают динамическую диспетчеризацию (runtime dispatch), что означает, что решение о том, какой метод вызвать, принимается во время выполнения программы, а не во время компиляции.

Пример виртуальной функции в C++:



В данном примере метод `Draw` является виртуальным в базовом классе `Shape`, и производный класс `Circle` переопределяет этот метод. В результате вызов `shapePtr->Draw()` в `main()` приводит к выполнению переопределенной версии метода `Draw` в классе `Circle`.

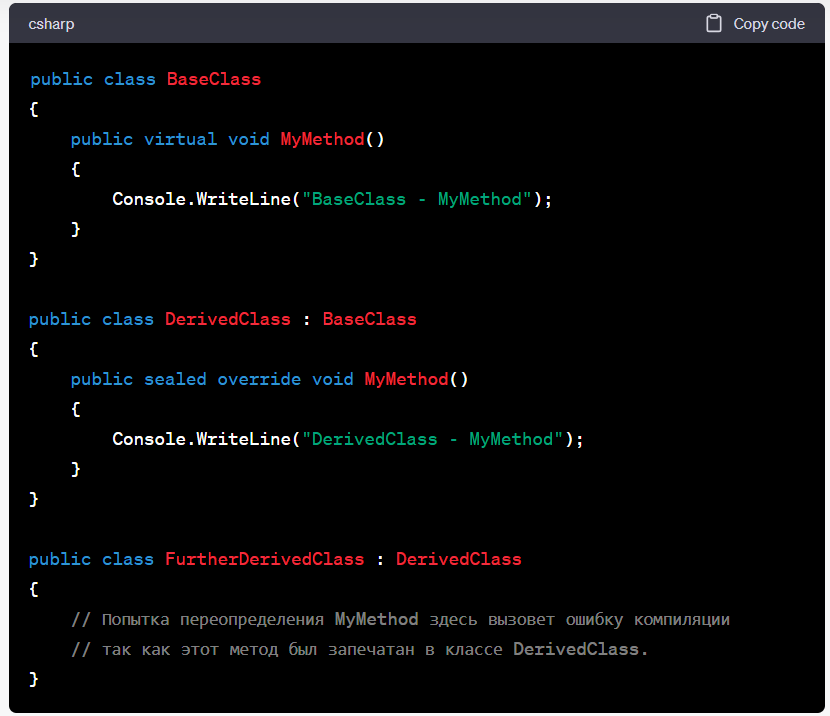
Правила переопределения

* 1) Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.
* 2) не может быть static или abstract
* 3) вызывается ближайший вариант, обнаруживаемый вверх по иерархии (многоуровневая)

**9. Как сделать запрет переопределения методов?**

В C# ключевое слово `sealed` используется для запрета переопределения методов в производных классах. Однако, если вы хотите разрешить наследование класса, но предотвратить переопределение конкретного метода, то можно сделать метод `sealed`, даже если сам класс не является `sealed`.

Пример:



В этом примере `MyMethod` в классе `DerivedClass` помечен как `sealed`, что предотвращает его переопределение в любом дальнейшем производном классе, таком как `FurtherDerivedClass`. Однако, `DerivedClass` сам по себе может быть использован в качестве базового класса для других классов.

JSON (JavaScript Object Notation) - это формат обмена данными, который предоставляет простой и читаемый способ представления структурированных данных. Он является текстовым форматом и легко читается как человеками, так и компьютерами.

Десериализация JSON - это процесс преобразования данных в формате JSON в объекты или структуры данных в программе.

`JObject` - это тип данных в библиотеке JSON.NET (Newtonsoft.Json) для работы с JSON-объектами в рамках языка C#.