**1. Назовите принципы ООП. Поясните каждый из них.**

Объектно-ориентированное программирование (ООП) - это парадигма программирования, которая базируется на следующих ключевых концепциях:

1. Классы - это шаблоны или чертежи, описывающие состояние (переменные-члены) и поведение (методы) объектов. Объекты - это конкретные экземпляры классов, созданные на основе их описания.

2. Инкапсуляция - это концепция, которая позволяет объединить данные и методы, работающие с этими данными, внутри класса и скрыть их от прямого доступа извне класса. То есть, данные и методы класса доступны только через определенный интерфейс класса. Это обеспечивает контролируемый доступ к данным и повышает безопасность кода.

3. Наследование позволяет создавать новые классы (подклассы или производные классы), на основе существующих (родительских или базовых классов). Подклассы наследуют свойства и методы родительских классов и могут добавлять или изменять их.

4. Полиморфизм позволяет объектам разных классов предоставлять общий интерфейс (например, методы с одинаковыми именами), но реализовывать их по-разному. Это позволяет использовать объекты разных классов с одинаковыми методами в общем контексте.

**2. Назовите класс .NET, от которого наследуются все классы.**

В .NET Framework и .NET Core, все классы наследуются от класса **System.Object**, который также известен как **object** (с маленькой буквы) в C#.

**3. Охарактеризуйте открытые методы System.Object.**

**Equals(object obj):** Сравнивает текущий объект с другим объектом и возвращает **true**, если они равны, и **false**, если они различны. Этот метод можно переопределить в пользовательских классах для более точного сравнения.

**GetHashCode():** Возвращает хеш-код текущего объекта, который используется для оптимизации хранения объектов в коллекциях, таких как **Dictionary** и **HashSet**.

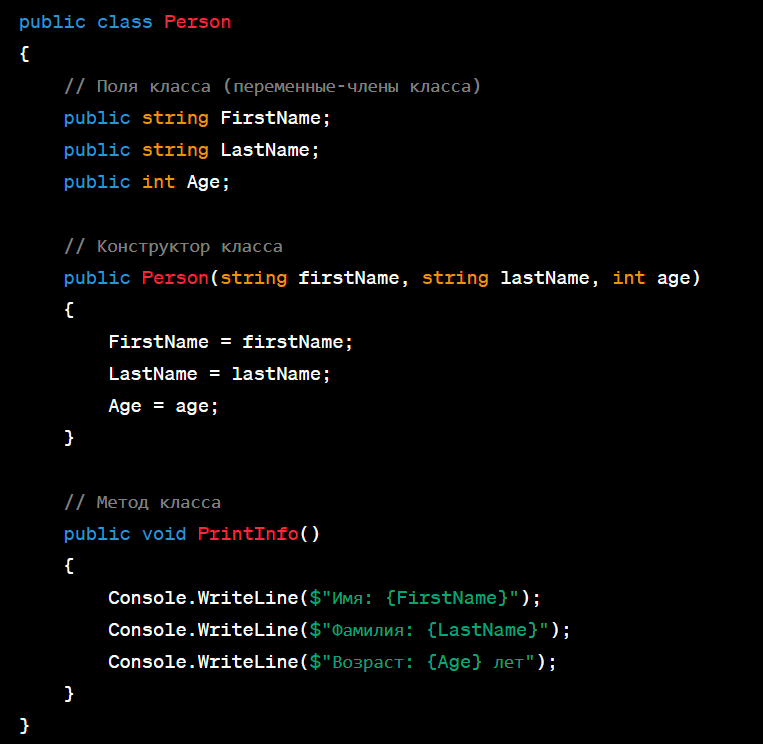
**GetType():** Возвращает тип (класс) текущего объекта в виде объекта **System.Type**, который предоставляет информацию о типе объекта.

**ToString():** Возвращает строковое представление текущего объекта. По умолчанию это полное имя типа объекта.

**4. Охарактеризуйте закрытые методы System.Object.**

Закрытых методов в классе **System.Object** нет. Все методы, перечисленные выше, являются открытыми и доступными для переопределения в производных классах.

**5. Приведите пример определения класса.**

****

**6. Какие ключевые слова можно использовать при определении класса?**

**class**: Определяет класс.

**partial**: Позволяет разделить определение класса на несколько частей в разных файлах.

**sealed**: Запрещает наследование от данного класса.

**abstract**: Определяет абстрактный класс, который не может иметь экземпляров и требует, чтобы его методы были реализованы в производных классах.

**static**: Обозначает статический класс, который не может иметь экземпляров и используется для группировки методов и свойств, не зависящих от конкретных экземпляров.

**internal**: Определяет класс с уровнем доступа, ограниченным текущей сборкой.

**7. В чем отличие между объектом и классом?**

Отличие между объектом и классом:

Класс - это шаблон или чертёж, описывающий структуру и поведение объектов. Это абстрактное описание, которое определяет поля (переменные) и методы (функции), которые будут у объектов этого класса.

Объект - это конкретный экземпляр класса, созданный на основе его описания. Объект имеет конкретные значения для полей, определённых в классе, и может вызывать методы, определённые в классе.

**8. Что такое конструктор? Когда вызывается конструктор?**

Конструкторы (constructors) в C# - это специальные методы класса, которые вызываются при создании нового объекта этого класса. Они используются для инициализации объектов, установки начальных значений полей класса и выполнения других операций, необходимых при создании экземпляра класса.

1. Конструктор имеет то же имя, что и класс, к которому он принадлежит. Он не имеет возвращаемого значения и может иметь различные сигнатуры, в зависимости от количества и типов параметров

2. Конструкторы могут принимать параметры, которые передаются при создании объекта. Эти параметры используются для инициализации полей объекта.

3. Конструкторы могут иметь параметры по умолчанию, что позволяет создавать объекты с разными наборами параметров или без них.

4. Если класс имеет наследование, конструкторы подкласса могут вызывать конструкторы базового класса с помощью ключевого слова `base`.

5. Класс может иметь статический конструктор, который вызывается один раз при первом обращении к классу или создании объекта этого класса.

Закрытый конструктор (private constructor) - это конструктор класса, который имеет модификатор доступа **private**, что означает, что он доступен только внутри самого класса и не может быть вызван извне класса.

**9. Перечислите свойства конструктора?**

конструкторы могут иметь параметры, которые используются для инициализации полей и свойств объекта при его создании. Эти параметры являются аргументами конструктора и могут различаться в зависимости от потребностей класса.

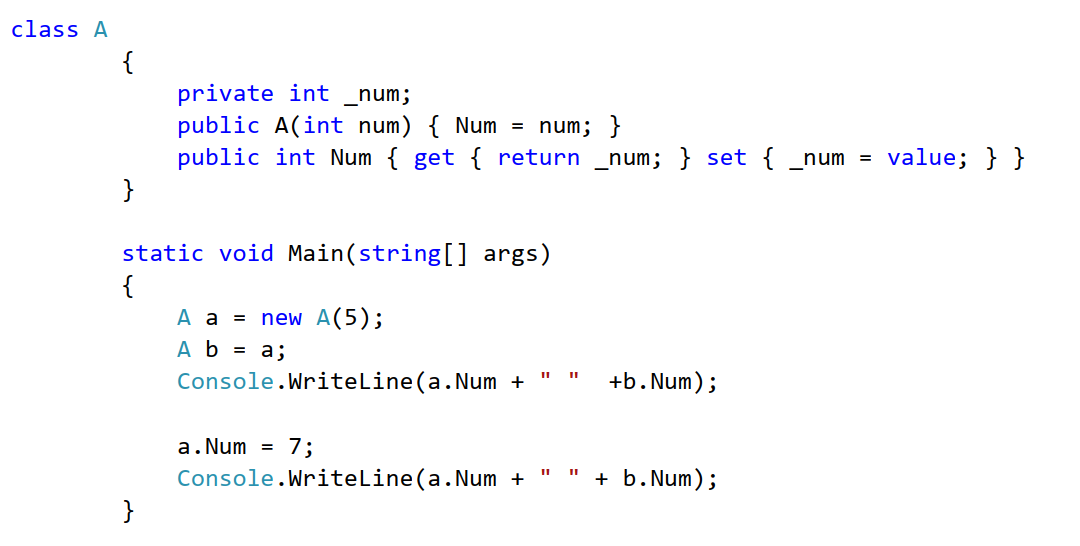
**10. Что такое деструктор (destructor) ?**

Вместо деструкторов в C# используется сборщик мусора (garbage collector), который автоматически уничтожает объекты, когда они больше не используются. Необходимость явно определять деструкторы отсутствует, и разработчику не нужно беспокоиться о высвобождении ресурсов.

**11. Что такое this?**

**this** - это ключевое слово в C#, которое ссылается на текущий экземпляр объекта внутри метода класса. **this** позволяет явно указать, что используется поле (или свойство) объекта, а не параметр метода с таким же именем.

**12. Что будет выведено в результате выполнения**

****

5 5 7 7

**13. Какие спецификаторы доступа для класса и методов класса существуют в C#?**

1. **public**: Класс и его члены доступны из любого места в программе и из других сборок.
2. **internal**: Класс и его члены доступны только в пределах текущей сборки (assembly). Это является значением по умолчанию, если спецификатор доступа не указан.
3. **protected internal**: Класс и его члены доступны как внутри текущей сборки, так и для производных классов (наследников) вне этой сборки.
4. **protected**: Класс и его члены доступны только для производных классов (наследников) внутри текущей сборки.
5. **private**: Класс и его члены доступны только в пределах текущего класса. Этот уровень доступа используется, когда нужно скрыть детали реализации от внешнего мира.

**14. Опишите модификатор protected internal.**

Модификатор доступа **protected internal** представляет собой комбинацию двух модификаторов доступа в C#: **protected** и **internal**. Этот модификатор предоставляет доступ к члену класса (например, методу, полю или свойству) или к самому классу изнутри текущей сборки (assembly) и также для производных классов (наследников) вне этой сборки.

Когда модификаторы **internal** и **protected** объединяются в **protected internal**, это означает, что члены с этим модификатором доступны как внутри текущей сборки, так и для производных классов (наследников) вне этой сборки. Это обеспечивает определенную гибкость в контроле доступа к членам класса, которые могут быть расширены и использованы за пределами текущей сборки, но только для наследников.

// Класс с модификатором protected internal

internal class MyBaseClass { protected internal void MyMethod()

{ // Этот метод доступен внутри текущей сборки и для производных классов } }

**15. Зачем и как используются ref и out параметры функции?**

`ref` и `out` - это ключевые слова в C#, используемые в параметрах методов для передачи аргументов по ссылке и возврата нескольких значений из метода. Они позволяют изменять значения переменных, переданных в метод, и возвращать несколько значений, соответственно. Вот их различия:

1. \*\*`ref`\*\*:

- Параметры, передаваемые по ссылке, должны быть инициализированы до вызова метода.

- Значение параметра может быть изменено внутри метода, и это изменение будет отражено на вызывающей стороне.

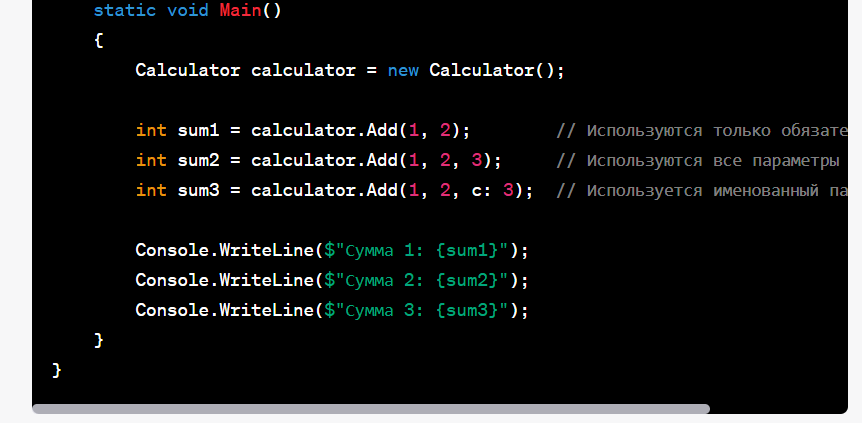
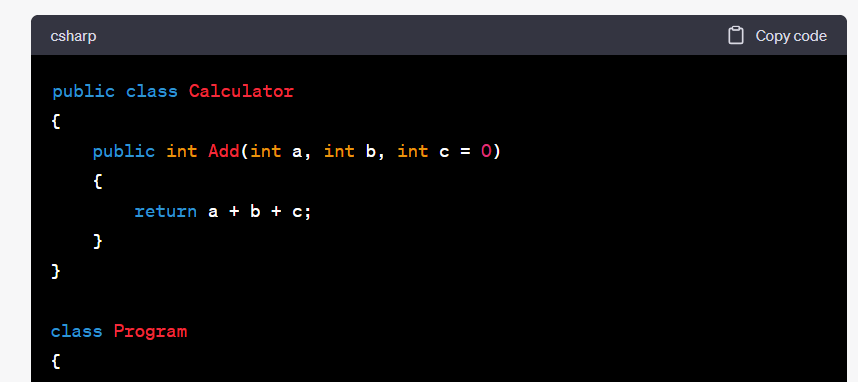
2. \*\*`out`\*\*:

- Параметры `out` не требуется инициализировать до вызова метода; они считаются неинициализированными до момента присвоения им значения внутри метода.

- Метод должен обязательно присвоить значение параметру `out` до его завершения.

- `out` используется, когда метод должен вернуть несколько значений.

**16. Приведите пример необязательных и именованных параметров метода.**



В этом примере метод **Add** имеет три параметра: **a**, **b** и **c**, причем параметр **c** имеет значение по умолчанию 0, что делает его необязательным. Вы можете вызывать метод **Add** с разными комбинациями параметров:

* **calculator.Add(1, 2)** вызывает метод с использованием только обязательных параметров, и **c** устанавливается в значение по умолчанию (0).
* **calculator.Add(1, 2, 3)** вызывает метод с использованием всех параметров, и **c** равно 3.
* **calculator.Add(1, 2, c: 3)** вызывает метод с использованием именованного параметра **c**, что делает код более читаемым и понятным.

**17. Приведите пример полей класса – статические, константные, только для чтения.**

1. **Статические поля (static)**: Статические поля принадлежат классу, а не экземпляру класса. Они общие для всех экземпляров класса и доступны через имя класса, а не через экземпляр.

public static int StaticField = 10;

1. **Константные поля (const)**: Константные поля имеют постоянное значение, которое нельзя изменить после компиляции. Они являются статическими по умолчанию и доступны через имя класса.

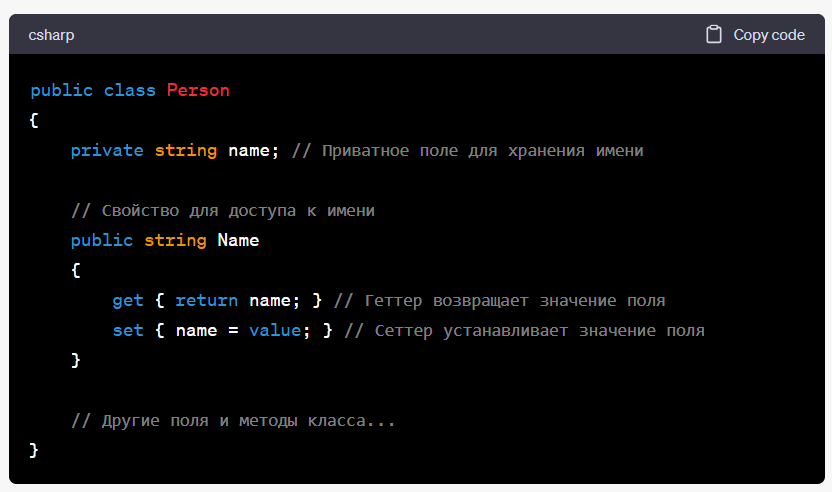
public const double Pi = 3.14159265359;

1. **Поля только для чтения (readonly)**: Поля только для чтения можно инициализировать только в конструкторе или при объявлении. Значение поля нельзя изменить после инициализации, и они могут иметь разные значения для каждого экземпляра класса.

public readonly int ReadOnlyField;

**18. Приведите пример определения свойств класса. Как свойства связаны с инкапуляцией?**

Вот пример определения свойств класса и их связи с инкапсуляцией:



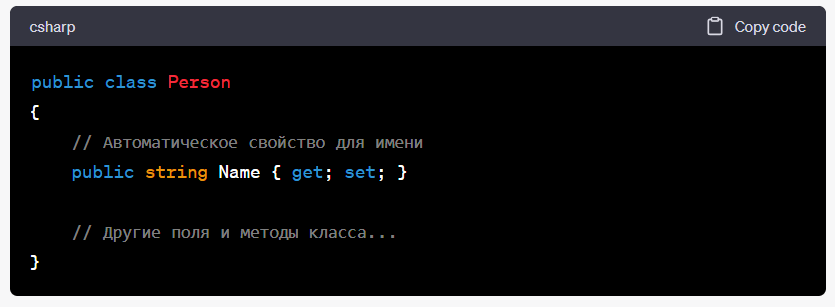
В данном примере у класса `Person` есть приватное поле `name`, предназначенное для хранения имени. Свойство `Name` позволяет получать (`get`) и устанавливать (`set`) это имя. Таким образом, свойство `Name` предоставляет интерфейс для работы с полем `name`, при этом оно может скрывать детали реализации, что является одним из аспектов инкапсуляции.

**19. Назовите явное имя параметра, передаваемого в метод set свойства класса?**

Явное имя параметра, передаваемого в метод `set` свойства класса, называется `value`. Это ключевое слово представляет передаваемое значение, которое будет установлено в свойство. Например, в предыдущем примере `set { name = value; }`, `value` представляет переданное значение, которое будет установлено в поле `name`.

**20. Что такое автоматические свойства?**

Автоматические свойства (auto-implemented properties) предоставляют упрощенный синтаксис для определения свойств класса без явного объявления поля. Вот пример:

В этом примере `Name` - это автоматическое свойство, которое не имеет явного поля для хранения данных. Компилятор C# автоматически создает скрытое приватное поле и реализацию геттера и сеттера для этого свойства. Автоматические свойства облегчают создание простых свойств без необходимости явного определения полей и методов.

**21. Что такое индексаторы класса? Какие ограничения существуют на  
индексатор?**

Индексаторы (indexers) в C# позволяют создавать классы, которые можно индексировать, как массивы. Они предоставляют способ доступа к элементам объекта класса с использованием индекса внутри квадратных скобок `[]`. Индексаторы позволяют обеспечивать доступ к элементам коллекции или данных в объекте класса с помощью удобного синтаксиса, подобного доступу к элементам массива.

Ограничения на индексаторы:

1. Индексатор должен быть определен внутри класса или интерфейса.
2. Индексаторы могут иметь один или несколько параметров индексации, но эти параметры должны быть значимыми типами данных (например, **int**, **string**, **enum**, **char**, и так далее).
3. Индексаторы могут иметь только **get** или только **set**, или оба метода.
4. Параметры индексации могут быть произвольными (например, **int**, **string**, **object**, и так далее), но они должны быть объявлены как параметры типа **value**.
5. Нельзя использовать ключевое слово **this** внутри индексатора в качестве имени параметра, так как оно зарезервировано для обращения к самому индексатору.

**22. Что такое перегруженный метод?**

Перегруженный метод (method overloading) - это концепция в объектно-ориентированном программировании, которая позволяет определить несколько методов с одним и тем же именем внутри класса или в рамках одного интерфейса, но с различными параметрами. Перегрузка методов позволяет использовать одно и то же имя метода для выполнения различных действий в зависимости от переданных аргументов.

**23. Что такое partial класс и какие его преимущества?**

Ключевое слово **partial** в C# используется для создания частичных (partial) классов. Частичный класс - это класс, который может быть разделен на несколько физических файлов. Это позволяет разработчикам разделять определение класса между несколькими файлами, при этом компилятор C# объединит их в один класс при компиляции.

**24. Что такое анонимный тип в C#?**

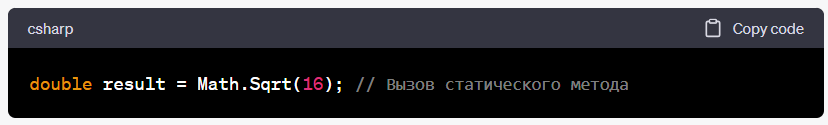
Анонимный тип (Anonymous Type) в C# - это специальный тип данных, который позволяет создавать объекты без явного определения класса. Анонимные типы часто используются для создания временных объектов с набором свойств для хранения данных.

Анонимные типы полезны, когда вам нужно быстро создать структуру данных для временного использования, например, при проектировании LINQ-запросов или передаче данных между частями программы.

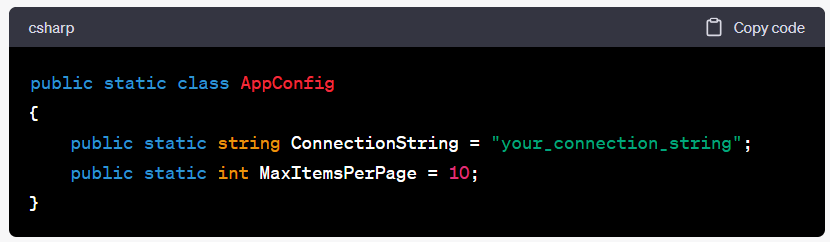
**25. Для чего делают статические классы?**

Статические классы в C# представляют собой классы, которые не могут быть инстанциированы (то есть, у них нельзя создать объект), и все их члены и методы являются статическими. Они имеют несколько специфических применений:

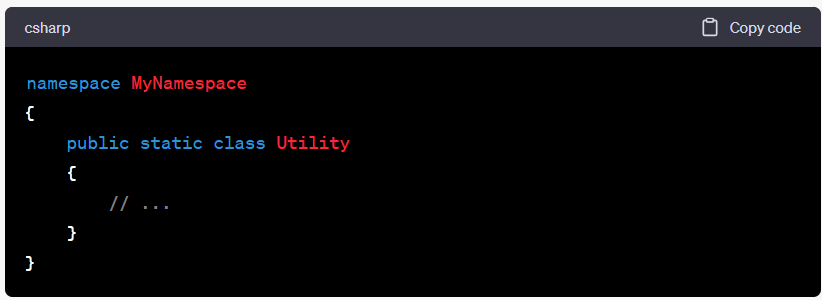
1. Статические классы часто используются для хранения методов, которые выполняют какие-либо утилитарные операции и не зависят от состояния объекта.



3. \*\*Хранение констант и настроек\*\*: Статические классы могут использоваться для хранения констант, настроек и параметров приложения, которые могут быть доступны из любой части кода.



5. \*\*Пространства имен\*\*: Статические классы могут использоваться для организации функциональности в пространствах имен без создания объектов.



Статические классы предоставляют удобный способ сгруппировать связанные методы и данные в одном месте, к которому можно обращаться без создания экземпляров классов.

**26. В чем отличие статического поля от экземплярного?**

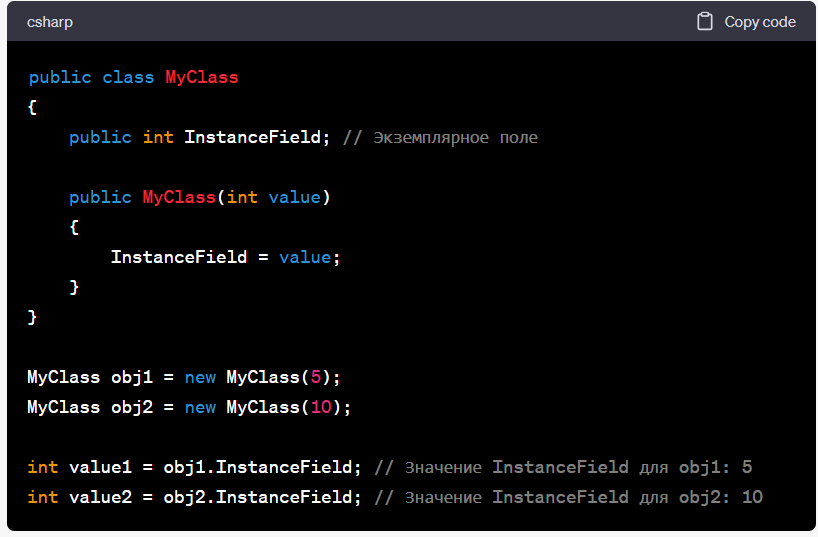
Отличие между статическими и экземплярными полями в C# заключается в их принадлежности к классу и способе доступа к данным:

1. \*\*Экземплярные поля (Instance Fields)\*\*:

- Принадлежат каждому конкретному экземпляру класса. Каждый объект этого класса имеет свою собственную копию экземплярных полей.

- Доступ к экземплярным полям осуществляется через объект этого класса.

- Каждый объект имеет свои собственные значения экземплярных полей.



2. \*\*Статические поля (Static Fields)\*\*:

- Принадлежат классу в целом, а не конкретным объектам этого класса. У всех объектов данного класса будет общая копия статических полей.

- Доступ к статическим полям осуществляется через имя класса, а не через объекты.

- Значение статических полей одинаково для всех объектов данного класса.



**27. Поясните работу статических конструкторов.**

Статический конструктор в C# - это специальный конструктор, который вызывается один раз при первом обращении к классу или при первом доступе к статическим членам класса. Он предназначен для инициализации статических полей или выполнения других действий, связанных с классом, до его использования.

Статический конструктор не имеет доступа к экземплярным членам класса, таким как экземплярные поля или методы. Его задачей является только инициализация статических данных.

**28. Какая разница между поверхностным (shallow) и глубоким (deep) копированием?**

1. \*\*Поверхностное (shallow) копирование\*\*:

- При поверхностном копировании создается новый объект, который является копией исходного объекта.

- Сами данные (поля) объекта копируются по ссылке. Это означает, что если исходный объект содержит ссылки на другие объекты, то новый объект также будет ссылаться на те же объекты, что и исходный.

- Изменения в данных, на которые ссылается копия, будут отражаться и в исходном объекте и наоборот.

2. \*\*Глубокое (deep) копирование\*\*:

- При глубоком копировании создается новый объект, и все данные внутри исходного объекта также копируются. Это включает в себя рекурсивное копирование всех объектов, на которые ссылаются данные внутри исходного объекта.

- Глубокое копирование создает полностью независимую копию объекта и всех объектов, на которые он ссылается. Изменения в новой копии не влияют на исходный объект и наоборот.

**29. В чем разница между равенством и тождеством объектов?**

1. \*\*Равенство объектов (Equality)\*\*:

- Равенство объектов означает, что два объекта имеют одинаковое состояние или одинаковое значение.

- Равенство определяется на основе содержимого объектов. Два объекта могут быть равными, даже если они разные по ссылке в памяти.

- В C# равенство объектов обычно определяется с помощью метода `Equals`, который может быть переопределен в пользовательских классах.

2. \*\*Тождество объектов (Identity)\*\*:

- Тождество объектов означает, что две ссылки указывают на один и тот же объект в памяти.

- Тождество можно проверить с помощью оператора `==`, который сравнивает ссылки на объекты, а не их содержимое.

- Два объекта могут быть тождественными, если они ссылаются на одну и ту же область памяти.

**30. Что такое частичные классы и частичные методы?**

\*\*Частичные классы (Partial Classes)\*\* и \*\*частичные методы (Partial Methods)\*\* - это концепции в C#, предназначенные для разделения определений классов и методов на несколько частей в разных файлах. Это удобно для организации кода в больших проектах и для совместной работы над классами между разными разработчиками.

1. \*\*Частичные классы (Partial Classes)\*\*:

- Частичные классы позволяют разделить определение одного класса на несколько фрагментов (части), которые могут быть определены в разных файлах.

- Эти части объединяются компилятором в один класс во время компиляции.

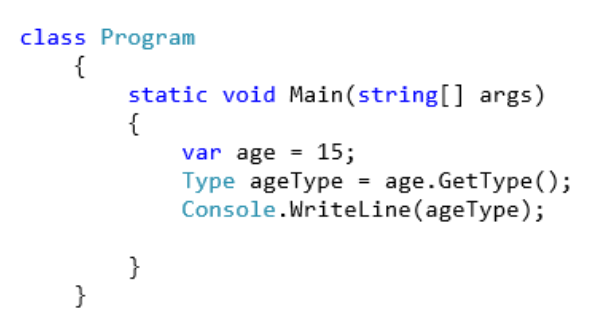
2. \*\*Частичные методы (Partial Methods)\*\*:

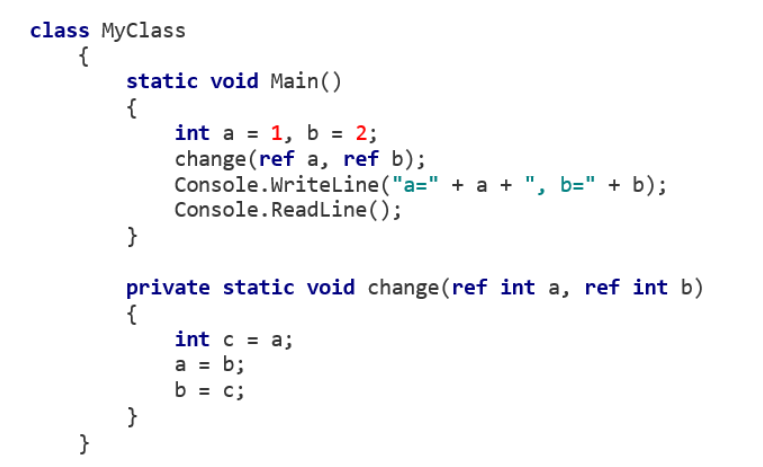
- Частичные методы - это методы, которые также могут быть разделены на определение и реализацию в разных частях частичного класса.

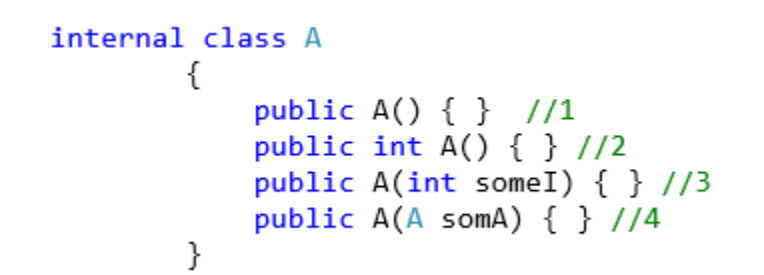
- Частичные методы могут быть определены только в частичных классах и обычно имеют пустое тело в одной части, и реализацию (или оставаться неопределенными) в другой части.

- Если частичный метод не имеет реализации в одной из частей, компилятор полностью исключает его из сборки, что может сэкономить ресурсы.

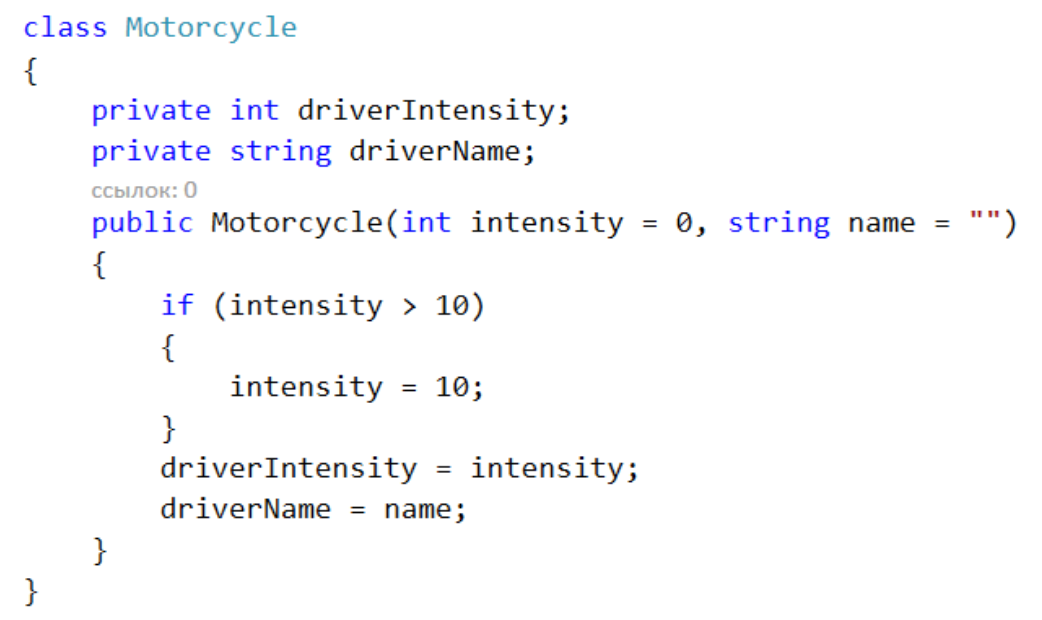
**31. Что будет выведено на консоль результате выполнения следующего кода:**

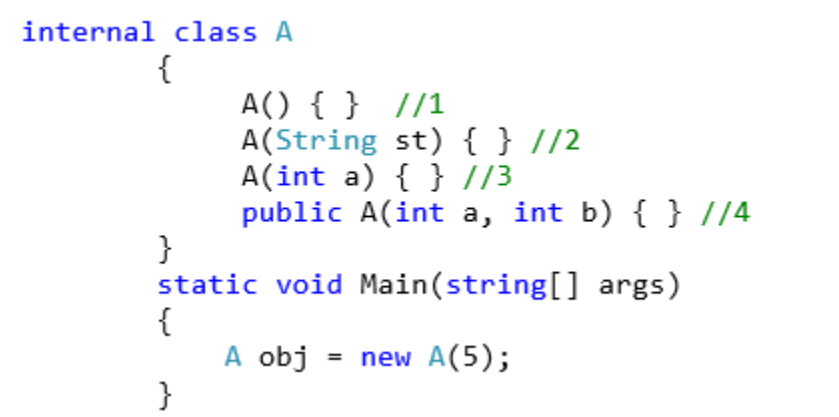
**  
32. Что будет выведено на консоль результате выполнения следующего кода:**

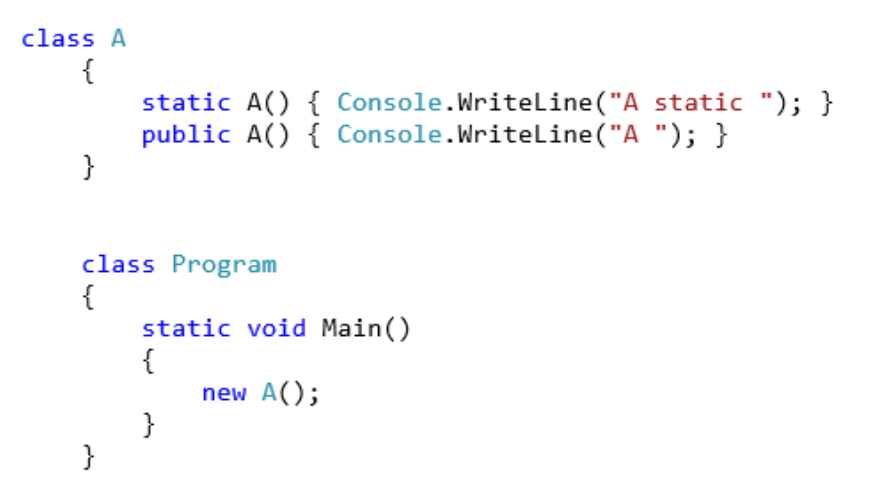
**  
33. Пусть задан следующий класс. Какой из конструкторов задан неверно?**

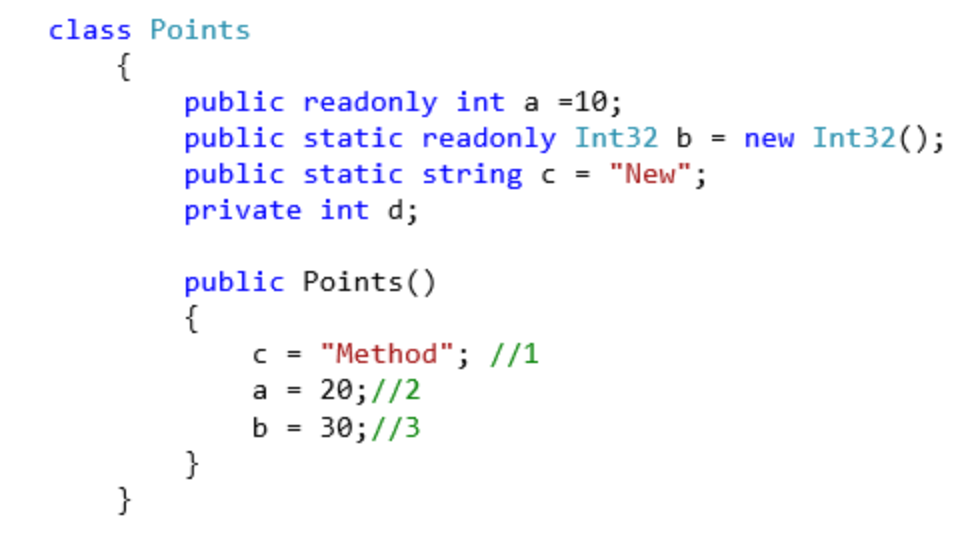
****

**34. Пусть задан следующий класс. Сколько аргументов может быть задано при вызове конструктора данного класса?**

**  
35. Почему не удается создать объект класса A?**

**  
36. Что будет выведено в консоль при выполнении данной программы?**

**  
37. Какая строка приведенного далее класса вызовет ошибку компиляции?**

****