1. Язык C#. Краткая история и характеристика языка. Типы данных и операторы в C#. Классы в C#. Пространства имен в C#.

С# – это объектно-ориентированный язык программирования. Он был создан в период с 1998 по 2002 год командой инженеров Microsoft под руководством Андерса Хейлсберга и Скотта Вильтаумота. Язык входит в семью С-подобных языков. Синтаксис приближен к Java и C++.

Его особенности:

• статистическая типизация,

• поддерживается полиморфизм,

• поддерживается перегрузка операторов,

• доступна делегация, атрибуты, события, обобщенные типы и анонимные функции.

**В** **языке** **C#** реализованы следующие базовые **типы** **данных**:

1. Целые типы: sbyte, byte, short, ushort, int, uint, long, ulong
2. Типы, в которых сохраняются данные с плавающей запятой (соответствуют вещественному типу): float, double, decimal
3. Символьный тип: char
4. Логический тип: bool
5. Строчный тип: string

**Операторы** — это символы, которые используются для выполнения операций с операндами. Операндами могут быть переменные и/или константы.

Операторы используются для управления переменными и их значениями в программе. C# поддерживает ряд операторов, разделенных по типам выполняемых ими операций.

Основной оператор присваивания = используется для присвоения значений переменным.

Арифметические операторы используются для выполнения арифметических операций: сложение, вычитание, умножение, деление и т. д.

Операторы сравнения (они же — операторы отношения) используются для проверки связи между двумя операндами. Если сравнение истинно, результат будет true, иначе — false. Операторы отношения обычно используются в условиях и циклах.  
 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Операторы инкремента | ++ | и декремента | -- | могут использоваться как |
| префикс и постфикс. Если использовать их как префикс, изменение значения | | | | |

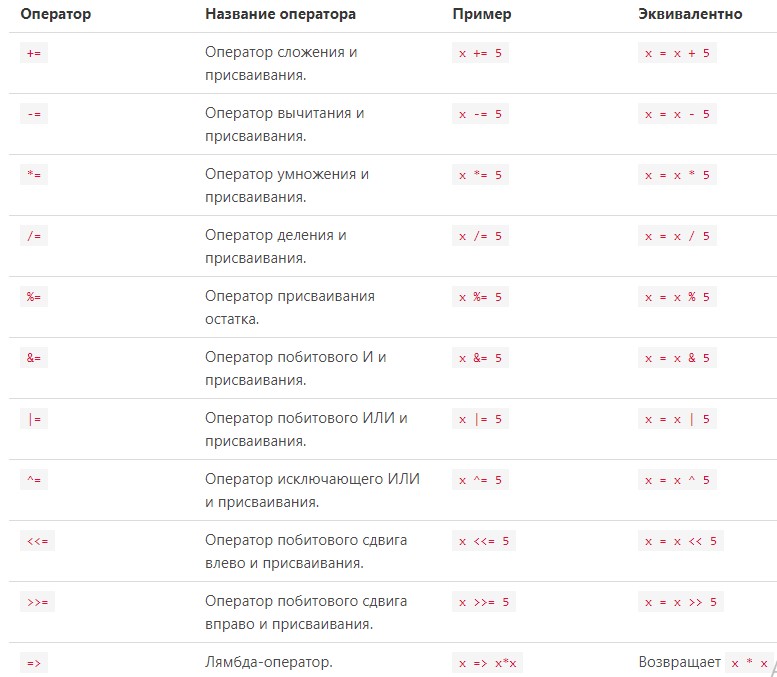
Логические операторы используются для выполнения таких логических операций, как AND (ИЛИ), OR (И) и т. д. Логические операторы работают с булевыми выражениями и возвращают булевые значения true и false. Логические операторы нужны для принятия решений и в циклах. Унарные операторы работают только с одним операндом — это отличает их от всех остальных операторов.   


переменной отображается в той же строке, а если как постфикс, изменение значения переменной отображается в следующей строке.

Побитовые операторы и операторы побитового сдвига используются для выполнения манипуляций с битами.



Составные операторы присваивания



**Класс** — это логическая структура, позволяющая создавать свои собственные пользовательские типы. Класс определяет данные и поведение типа.Классы объявляются с помощью ключевого слова class.

Внутри себя, класс может содержать **методы**, **поля** и **свойства**.

**Методы** похожи на функции из языков группы структурного программирования. Фактически они определяют то, как можно работать с данным классом или объектами класса. **Поля** – это переменные, связанные с данным классом, а **свойства** – это конструкции специального вида, которые упрощают работу с полями

**Пространства имен в C#** определяет область кода, в которой гарантируется уникальность всех идентификаторов. Это своеобразные контейнеры, в которых находятся другие пространства имен, классы и т. д.

В C# определить пространство имен можно с помощью ключевого слова namespace.

*namespace имя\_пространства\_имен*

*{*

*// Тело пространства имен*

*}*

Например: *namespace MyNamespace*

*{*

*class MyClass*

*{*

*public void MyMethod()*

*{*

*System.Console.WriteLine("Создаем пространство имен");*

*}*

*}*

*}*

В приведенном выше примере мы создали пространство имен с именем MyNamespace. Оно состоит из класса MyClass. MyMethod в свою очередь — метод класса MyClass.

1. Свойства и индексаторы в языке C#. Преимущества и недостатки данных элементов языка. Унифицированная система типов в языке C#.

Свойства обеспечивают простой доступ к полям классов и структур, узнать их значение или выполнить их установку.

|  |
| --- |
| [модификаторы] тип\_свойства название\_свойства  {      get { действия, выполняемые при получении значения свойства}      set { действия, выполняемые при установке значения свойства}  } |

Вначале определения свойства могут идти различные модификаторы, в частности, модификаторы доступа. Затем указывается тип свойства, после которого идет название свойства. Полное определение свойства содержит два блока: **get** и **set**.

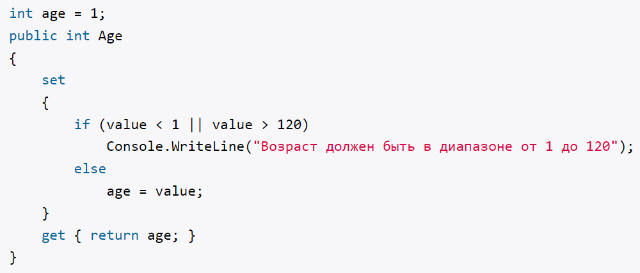
В блоке **get** выполняются действия по получению значения свойства. В этом блоке с помощью оператора **return** возвращаем некоторое значение.

В блоке **set** устанавливается значение свойства. В этом блоке с помощью параметра **value** мы можем получить значение, которое передано свойству.

В программе мы можем обращаться к этому свойству, как к обычному полю. Если мы ему присваиваем какое-нибудь значение, то срабатывает блок **set**, а передаваемое значение передается в параметр **value**.

Если мы получаем значение свойства, то срабатывает блок **get**, который по сути возвращает значение переменной.

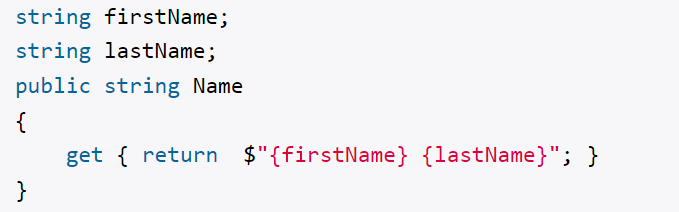
Свойства позволяют вложить дополнительную логику, которая может быть необходима при установке или получении значения.



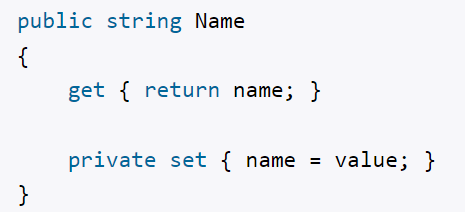
Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяет только блок **get**, то такое свойство доступно только **для чтения** - мы можем получить его значение, но не установить.

И, наоборот, если свойство имеет только блок **set**, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить:

Свойства необзательно связаны с определенной переменной. Они могут вычисляться на основе различных выражений



Мы можем применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам get и set.



**Индексаторы** позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

возвращаемый\_тип this [Тип параметр1, ...]

{

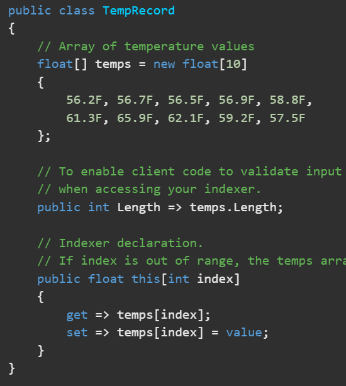
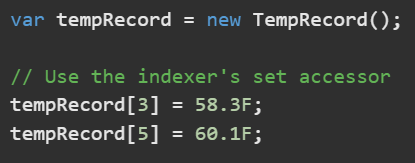
    get { ... }

    set { ... }

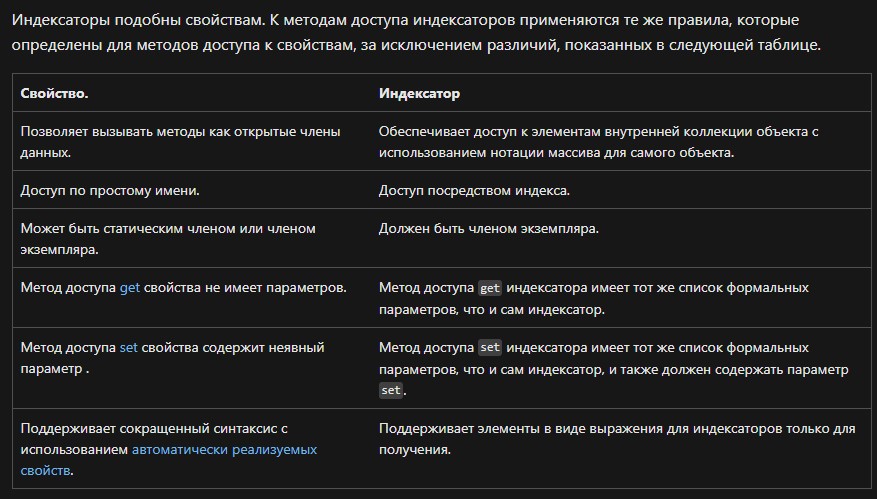
}

В отличие от свойств индексатор не имеет названия. Вместо него указывается ключевое слово **this**, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

Применение индексаторов упрощает работу с синтаксисом, позволяя создавать классы, структуры и интерфейсы, к которым клиентские приложения могут обращаться так же, как к массиву. В этом случае компилятор создает свойство Item (или свойство с другим именем, если присутствует IndexerNameAttribute) и соответствующие методы доступа. Индексаторы чаще всего реализуются в типах, предназначенных преимущественно для инкапсуляции внутренней коллекции или массива. Допустим, у вас есть класс TempRecord, представляющий журнал с 10 измерениями температуры по шкале Фаренгейта за период в 24 часа. Этот класс содержит массив temps типа float[] для хранения значений температуры. Реализация индексатора в этом классе позволит клиентам получать доступ к значениям температуры в экземпляре TempRecord, используя float temp = tempRecord[4] вместо float temp = tempRecord.temps[4].

Некоторые средства языка С# являются избыточными - в том смысле, что не добавляют к языку никаких функциональных возможностей, а лишь позволяют в иной форме записать то, что и так может быть выражено достаточно просто. К числу таких средств относятся свойства (properties) и индексаторы (indexers). Такие конструкции языка С#, как свойства, индексаторы, приводят к ситуациям, когда по тексту программы (программной единицы) невозможно понять природу используемых в ней объектов. Свойства могут быть неотличимы от полей, индексируемые объекты - от массивов.



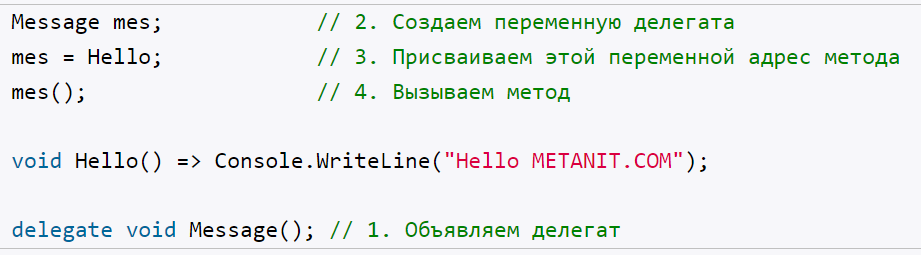
**Унифицированная система типов C#** фактически означает, что типы значений обрабатываются как object ссылки "по запросу". Из-за унификации библиотеки общего назначения, использующие тип object , могут использоваться со всеми типами, производными от object , включая ссылочные типы и типы значений.

1. Делегаты в языке C#. Модель событий в языке С#.

Делегаты представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Для объявления делегата используется ключевое слово **delegate**, после которого идет возвращаемый тип, название и параметры.

delegate void Message();

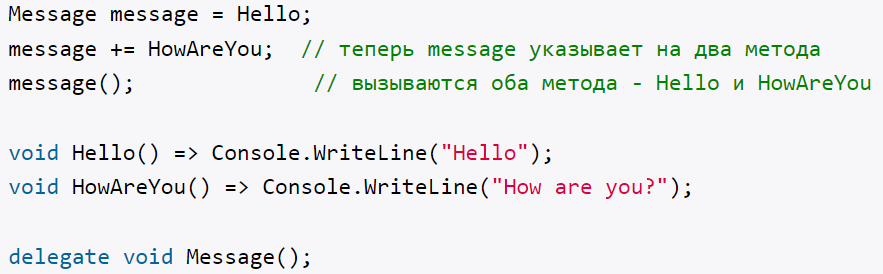


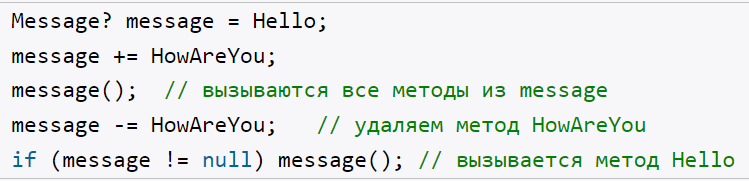
Делегаты могут принимать параметры и возвращать значения.

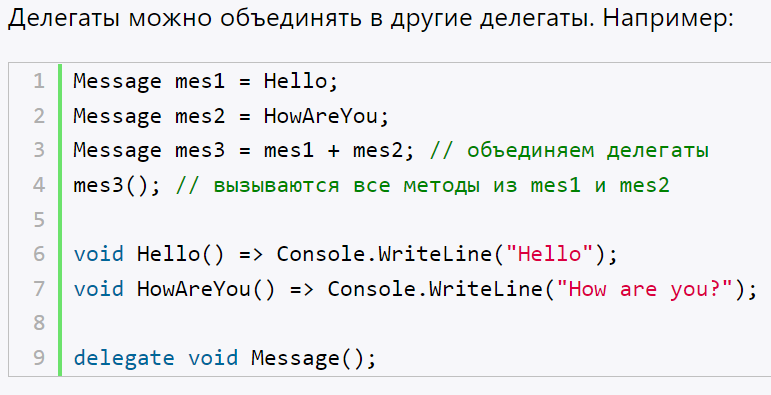
delegate int Operation(int x, int y);

Методы соответствуют делегату, если они имеют один и тот же возвращаемый тип и один и тот же набор параметров.

Для добавления методов в делегат применяется операция **+=**



Подобным образом мы можем удалять методы из делегата с помощью операций **-=**. 



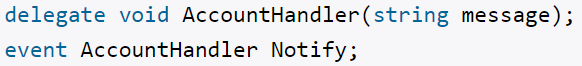
Другой способ вызова делегата представляет метод **Invoke()**:

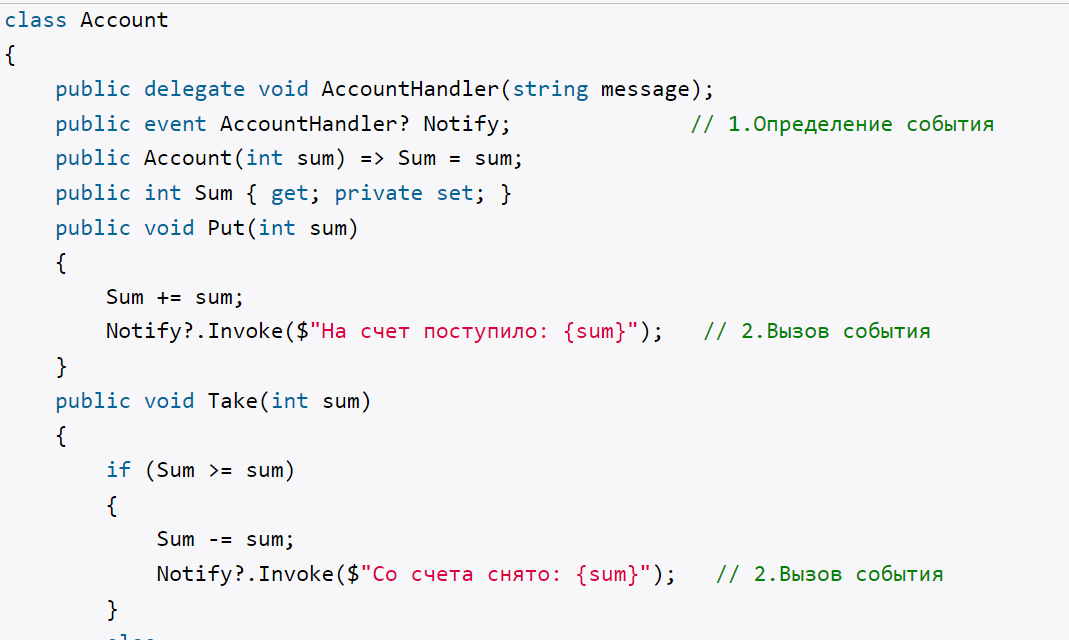
mes.Invoke();

Делегаты могут быть параметрами методов. Также делегаты можно возвращать из методов.

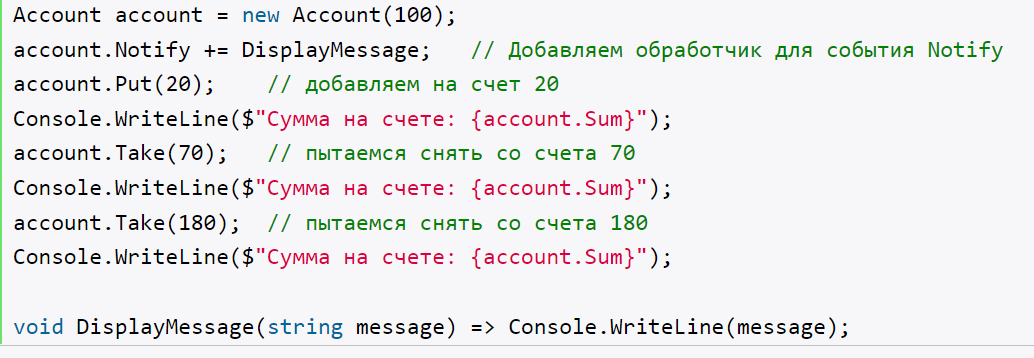
**События** сигнализируют системе о том, что произошло определенное действие.

События объявляются в классе с помощью ключевого слова **event**, после которого указывается тип делегата, который представляет событие.

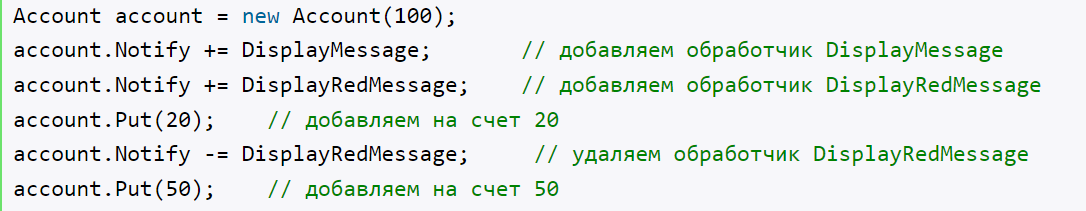




С событием может быть связан один или несколько обработчиков. Обработчики событий - это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются методы. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие. Для добавления обработчика события применяется операция **+=**:



Для одного события можно установить несколько обработчиков и потом в любой момент времени их удалить. Для удаления обработчиков применяется операция **-=**.

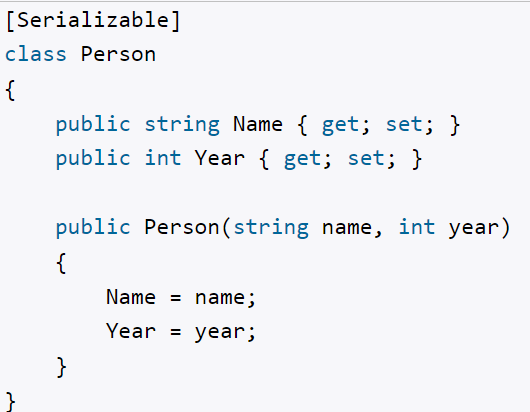


В качестве обработчиков могут использоваться не только обычные методы, но также делегаты, анонимные методы и лямбда-выражения.

1. Сериализация в языке С#. Стандартная и пользовательская сериализация.

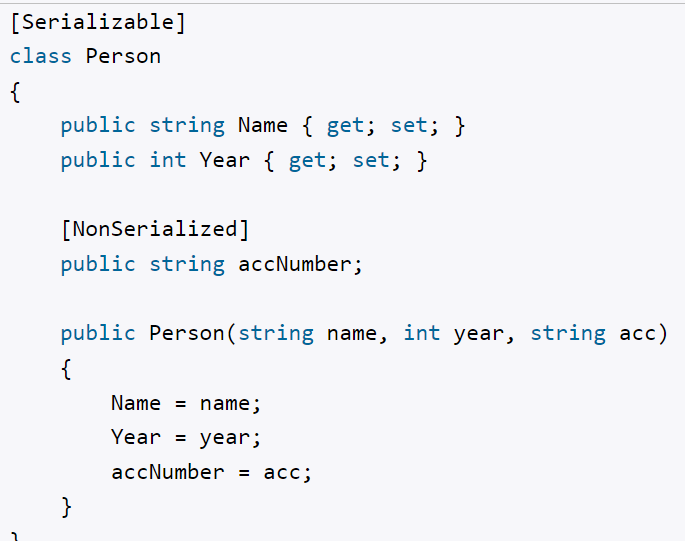
**Сериализация** представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - **десериализацию**, то есть получить из потока байтов ранее сохраненный объект.

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом **Serializable**



При отстутствии данного атрибута объект Person не сможет быть сериализован, и при попытке сериализации будет выброшено исключение SerializationException.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом **NonSerialized**:



При наследовании подобного класса, следует учитывать, что атрибут Serializable автоматически не наследуется. И если мы хотим, чтобы производный класс также мог бы быть сериализован, то опять же мы применяем к нему атрибут

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. Итак, в .NET можно использовать следующие форматы:

* бинарный
* SOAP
* xml
* JSON

Для каждого формата предусмотрен свой класс: для сериализации в бинарный формат - класс **BinaryFormatter**, для формата SOAP - класс **SoapFormatter**, для xml - **XmlSerializer**, для json - **DataContractJsonSerializer**.

Используя пользовательскую сериализацию, вы можете точно указать, какие объекты и как будут сериализованы. Класс должен иметь отметку SerializableAttribute и реализовывать интерфейс ISerializable. Если вы хотите настраивать и десериализацию объекта, следует использовать пользовательский конструктор.

Базовая сериализация использует платформу .NET для автоматической сериализации объекта. Единственное требование заключается в необходимости применить атрибут SerializableAttribute к классу. Атрибут NonSerializedAttribute также можно использовать для исключения из сериализации определенных полей.

При использовании базовой сериализации управление версиями объектов может привести к ошибкам. Если проблемы с управлением версиями важны, рекомендуется использовать пользовательскую сериализацию. Базовая сериализация является самым простым способом сериализации, но не дает почти никакого контроля над процессом.