Подробный синтаксис

# Основы типов

Члены класса делятся на 2 больших типа – поля и методы. Поля – члены класса, которые определяют состояние объекта или всего типа, если поле статическое. Методы определяют некоторые действия над объектами и/или всем типом. И поля, и методы имеют свои разновидности.

Сами типы могут иметь некоторые модификаторы. Все эти модификаторы существуют только в .NET, в более ранних версиях некоторые из модификаторов могут отсутствовать.

* *public* – означает, что класс может быть доступен за пределами данной сборки. Не ставьте сразу этот модификатор, если не уверены, что класс понадобиться за пределами сборки.
* *abstract* – означает, что класс имеет абстрактные методы, то есть методы, чьи реализации должны быть в наследуемых классах.
* *sealed* – запрещает наследоваться от класса. Полезно для работы виртуальных методов, поэтому всегда помечайте класс как *sealed*, если только точно не знаете, что от него будут наследоваться. Если вдруг выяснится, что наследование нужно, всегда можно будет убрать этот модификатор.
* *static* – *abstract* + *sealed*. Такие классы нужны, если вы хотите просто объединить набор статичных функций в один ряд. Например, существует класс *Math*, который просто содержит в себе ряд стандартных математических функций.
* *partial* – означает, что здесь только часть всей реализации класса. Например, все классы визуальных формы в Windows Forms и ASP.NET разделяются на часть, автоматически сгенерированную визуальным редактором, и часть, реализованную программистом.

Общие рекомендации следующие – всегда помечайте класс как *sealed*, если только точно не знаете, что от него будут наследоваться.

# Поля.

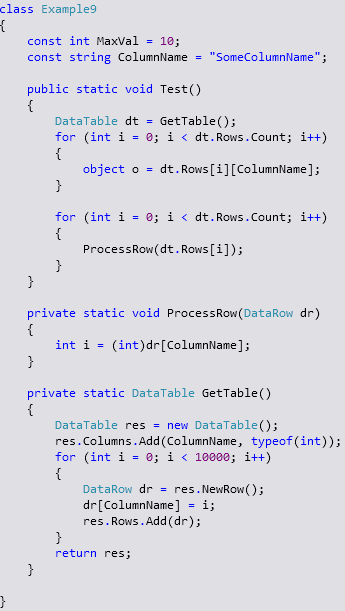
# Константы.



Статические неизменяемые поля. Значение можно задать только сразу после объявления константы. Реальные значения можно задать только примитивным типам, например, строке, *int* и прочим, но можно задать null и другим типам. Значения констант содержатся в метаданных. При использовании константы в коде компилятор заменяет сам код на значение из метаданных. Таким образом, фактически никаких переменных не создается, и память под них не выделяется. Соответственно, нельзя передавать константы в качестве параметров по ссылке и получить их адрес.

Однако здесь есть важная проблема – если константа, объявленная в одной сборке (назовем сборка 1) используется в другой сборке (назовем сборка 2), то при изменении значения константы в сборке 1 без перекомпиляции сборки 2 (например, подложив новую dll), значение константы в модуле 2 останется старым.

Тот факт, что данные констант не хранятся постоянно в памяти, а создаются при использовании – палка о двух концах. С одной стороны, это, безусловно, благо. В противном случае все константы, однажды объявленные, постоянно хранились бы в памяти. С другой стороны, со строковыми константами это может вылиться в не очень приятный эффект.



Рассмотрим функцию *Test*. В начале она получает большую таблицу с 10000 строками. В реальной задаче такая ситуация вполне возможна. Дальше циклически она обращается к колонке каждой строки, используя константу. В таком случае объект строки будет создан 10000 раз вместо того, чтобы создать один и использовать его каждый раз. Впрочем, велика вероятность, что JIT-компилятор оптимизирует эту часть кода так, чтобы использовался только один объект строки. Однако он точно не сможет оптимизировать другой вариант обработки, когда обращение к константе происходит в другой функции, поскольку JIT компилирует только одну функцию.

Таким образом, если какая-то строковая константа используется очень часто, то лучше использовать *static readonly*, для программиста их использование будет идентичным, но внутри всегда будет создаваться и использоваться только один объект строки.

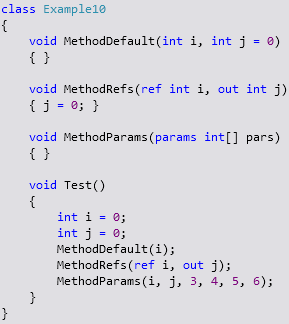
# readonly.

Модификатор поля, означающий, что поле не может быть изменено. Выставляется значение только в конструкторе или сразу при объявлении. Модификатор можно применять и для статических полей. В этом случае это будет похоже на константу, но значение будет храниться в объекте типа. Кроме того, можно задавать значения не только примитивным типам.

Необходимо иметь в виду, что если *readonly* объявлено поле Reference Type, то неизменным будет сама ссылка, а объект, на который ссылается поле.

# Методы.

# Параметры.



В *MethodDefault* 2 параметр необязательный. Если вызвать метод, используя только 1 параметр, то в функции будет использоваться значение по умолчанию (0).

В *MethodRefs* используются параметры по ссылке, то есть изменения значений параметров в функции приведут к изменениям самих значений переменный. Для IL *ref* и *out* идентичны. Отличия только для компилятора. Он требует, чтобы переменная, использующая *ref*, была инициализирована до вызова функции, а переменная, использующая *out*, – во время выполнения функции.

В *MethodParams* используется переменное число параметров. В этом случае создает статичный массив элементов, который и использует функция.

# Конструктор.

Метод, вызываемый при создании объекта. Его задача – инициализировать объект в некое начальное, но единое, т.е. рабочее состояние. Перед началом работы конструктора все поля объекта обнуляются. После этого происходит присваивание значений полей. И уже после этого вызывается конструктор. Можно определить несколько конструкторов с различным числом параметров, а также с разной видимостью. Если ни одного конструктора не объявлено, то автоматически создается публичный конструктор без параметров, называется «конструктор по умолчанию».

Конструктор для Value Type работает несколько иначе. Для Value Type всегда существует конструктор по умолчанию, его нельзя убрать, понизить видимость или изменить.

# Статичный конструктор.

Статичный конструктор – инициализирующий объект типа.

Здесь ключевой вопрос – когда вызывается этот метод. В CLR есть 2 различных варианта, когда вызывать его:

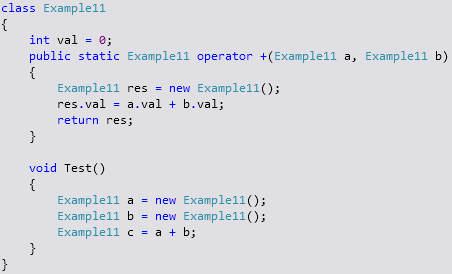
1. Точная семантика (precise semantics). Конструктор вызывается прямо перед созданием первого экземпляра объекта или прямо перед тем, как код обратится к полям класса. Т.е. использование такого метода позволяет вызывать конструктор в нужное время. Однако он значительно медленнее второго.
2. Перед-инициализацией-поля (before-field-init semantics). Конструктор вызывается за некоторое время до первого доступа к статичным полям или вызова методов (любых), или вызова конструктора экземпляра объекта. В этом случае конструктор может вызваться задолго до нужного времени.

IL-язык позволяет программно задавать, какой механизм использовать для статичного конструктора. Однако C# этой возможности не предоставляет, делая выбор самостоятельно. Выбор делает простой – если статичный конструктор объявлен прямо, то используется precise semantics, если только присваивание статичных полей после их объявления (это фактически тоже часть конструктора), то – before-field-init semantics. Поскольку before-field-init semantics более быстрый метод, то рекомендуется не использовать прямое использование конструктора без необходимости.

Чтобы избежать многократного вызова статического конструктора, в объекте типа есть флаг вызова конструктора. Когда первая нить приходит вызвать его, метод блокируется от других нитей, позволяя первой нити выполнить конструктор в одиночку. Когда метод разблокируется, остальные нити уже увидят, что конструктор вызывался.

Отдельно, нужно упомянуть, что статический конструктор для Value Type вызывается не всегда, поэтому лучше всего его не определять совсем.

# Переопределение операторов.



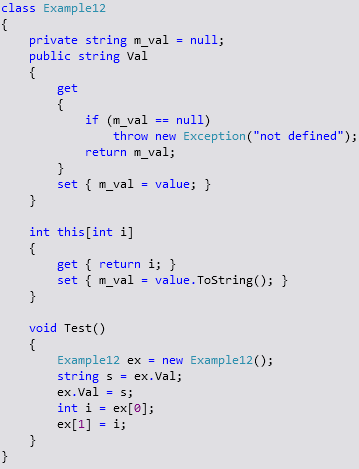
Синтаксически, чтобы переопределить оператор, необходимо создать новый *public static* метод с 1 или 2 параметрами (соответственно оператору), у которого вместо названия будет стоять *operator*, а после него сам переопределяемый оператор. C# также требует, чтобы один из параметров был того же типа, что и класс, в котором объявляется переопределение.

В IL-языке нет никаких операторов, в действительности компилятор C# переводит операторы в соответствующие методы со стандартными названиями. Например, бинарному оператору + соответствует метод *op\_Addition*. Также в метаданных методу присваивается флаг *specialname*.

Еще отмечу, что целочисленные типы (byte, Int32, Int64 и тд) не имеют таких методов, хотя все операции для них доступны. Вместо этого в IL-код сразу пишутся арифметические инструкции.

# Свойства.

Свойства – удобный с точки зрения синтаксиса способ использования getter и setter. Одной из основных идей ООП является то, что все поля должны быть закрытыми для внешних пользователей, а все действия над ними, как минимум должны осуществляться через getter и setter. Свойства дают предоставляют такой их вид, что синтаксически похоже на обращение к полям.



Здесь свойство *Val* обеспечивает доступ в полю m\_*val*, причем getter также осуществляет дополнительную проверку. Впрочем, свойства могут также предоставлять информацию о таком типе состояния объекта, за который напрямую не отвечает ни одно поле. Здесь представлен особый тип свойств – indexer или индексатор. Это свойство еще можно назвать переопределением скобок [].

При определении свойств фактически в метаданных создаются 1 или 2 метода – getter и/или setter (один из них можно не создавать), а также метка о самом свойстве.

Главная рекомендация относительно свойств – всегда создавайте их именно как надстройка над полем или именно свойства объекта. Ни в getter, ни в setter не должно быть сложной логики по получению чего-либо или совершения сложной операции.

Кроме того, getter никак не должен менять состояние объекта. Здесь есть важная особенность – в дебаге Visual Studio часто считывает значение getter' ов различных свойств, поэтому если вы пренебрежете этим советом, в дебаге могут происходить неожиданные эффекты.

JIT-компилятор оптимизирует работу свойств, делая вызовы getter и setter inline. Т.е. вписывает сам код метода в набор инструкций вместо того, чтобы создавать новую функцию с ее вызовом. Так он действует только в релизе, а также для небольших getter и setter.