Введение в .NET

# Основные понятия и принципы.

Прежде всего .NET – не язык программирования, это платформа.

Программы работают в специальной среде выполнения программ CLR (Common Language Runtime). При компиляции программы генерируется промежуточный IL(Instruction List)-код. Это т.н. «высокоуровневый объектно-ориентированный ассемблер». Он содержит инструкции, как ассемблер, но в то же время перед исполнением необходимо скомпилировать его в машинный код. Компиляция происходит на лету – во время работы программы. Но это именно компиляция, а не интерпретация. Программы для .NET могут быть написаны на различных поддерживаемых языках, так что можно написать отдельные модули на более подходящем задаче языке. Но это редко используется. Также есть возможность взаимодействовать с кодом, написанным на других, не .NET, языках.

# Portable Executable.

При компиляции программы на выходе получается PE-файл (Portable executable). Этот PE-файл является исполняемым для любых систем, в которых есть нужная версия .NET или Mono.

Кратко, что содержит PE-файл:

|  |  |
| --- | --- |
| PE32 или PE32+ header | PE32 может выполняться на 32- и 64- битных версиях Windows, PE32+ - только на 64-битной. |
| CLR-header | Содержит информацию, необходимую для CLR, например, поддерживаемую версию CLR, различные флаги, точку входа и т.д. |
| Метаданные | Наборы таблиц, которые содержат описания всех объявленных типов и их членов, а также описания типов, на которые ссылается модуль. Содержат и другие данные, например, все использованные в модуле строковые константы. |
| IL-код | Сам программный код. |

# JIT-компиляция.

Исполняемый IL-код компилируется по мере необходимости. При первом вызове любой функции она компилируется, стараясь оптимизировать полученный машинный код с учетом текущей системы. Полученный машинный код хранится в памяти, и при повторном вызове функции используется уже откомпилированный код.

Существует утилита NGen, которая компилирует PE-файл в машинный код и сохраняет результат в специальное хранилище. Этот код будет оптимизирован под текущую систему, поэтому не сможет работать на другой системе или после изменений в «железе».

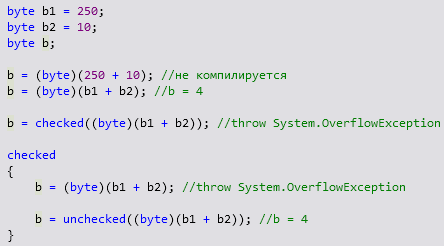
При вызове PE-файла сначала проверяется наличие такого сгенерированного файла, и если он есть, то он и запускается. Будет работать уже скомпилированный код, нет нужды в JIT. Однако полученный при такой компиляции код не будет иметь оптимизацию под текущее состояние самой среды выполнения. Например, при JIT-компиляции компилятору известны адреса некоторых констант или других статических полей, и он использует этот адрес напрямую. При компиляции с помощью NGen, эти данные неизвестны, поэтому компилятор использует другой, более медленный механизм.

Таким образом, при единственном достоинстве использования NGen – сокращение времени на первый вызов функций – есть много недостатков. В результате программы, скомпилированные с помощью NGen, работают медленнее, чем оригинальные PE-файлы. Поэтому в использовании NGen особой нужды нет, особенно для серверных программ, работающих продолжительное время.

# Основные операции.

Арифметические целочисленные операции проводятся только в 32- или 64-битных значениях. Поэтому если арифметические операции проводятся между, например, переменными типа байт, сначала переменные преобразуются в int, потом производится операция, а результат возвращается в виде int. Таким образом, нет смысла экономить на байтах, если для этих переменных предусматриваются арифметических операции.

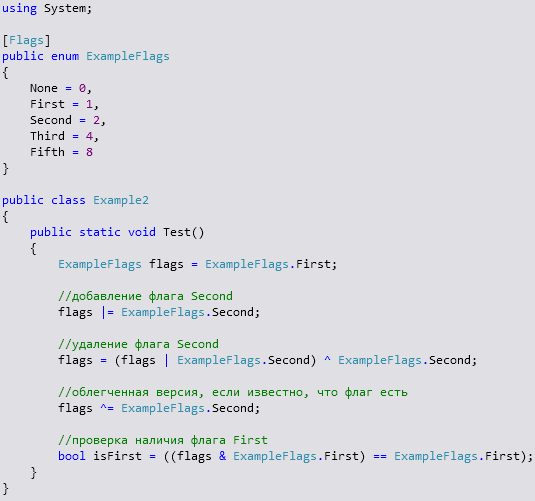
По умолчанию не проводится проверка на переполнение. Это может быть полезным при подсчете checksum и прочих подобных задачах. Чтобы включить проверку на переполнение в C# есть 2 оператора *checked* и *unchecked.*



Кроме того, можно задать эти параметры для всего проекта (Свойства проекта -> build -> advanced -> Check for arithmetic overflow/underflow). Использование операторов определяет, какие версии команд IL-языка будут использоваться (например, add или add.ovf). Операции без проверки на переполнение исполняются быстрее. Поэтому рекомендуется в дебажной сборке выставлять проверку на переполнение, чтобы обнаружить ошибку, а в релизной сборке убирать эту проверку. В случае, если программа не выполняет каких-либо критичных по времени операций, то эту проверку рекомендуется выставлять всегда.

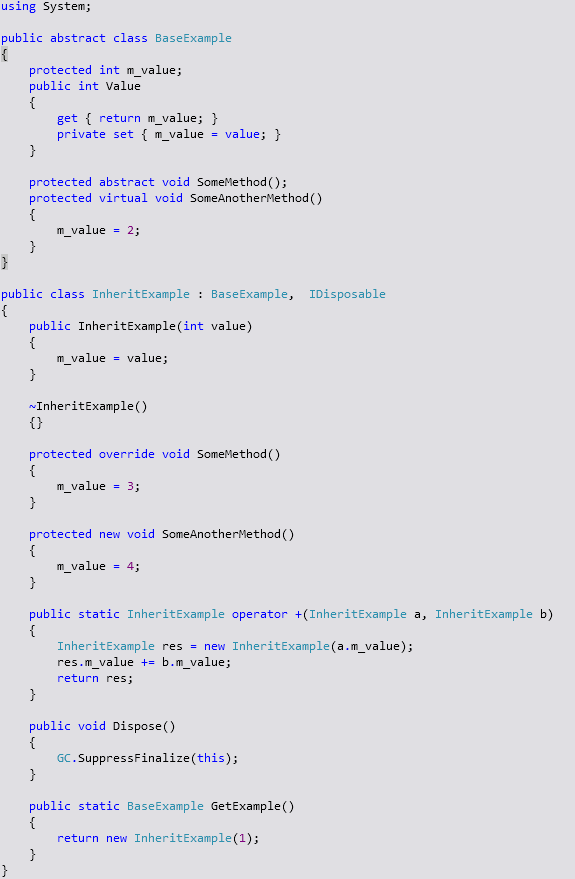
Бинарные операции:

1. Добавление флага – *set |= flag*
2. Удаление флага – *set = (set | flag ) ^ flag*
3. Удаление флага, если известно его наличие – *set ^= flag*
4. Проверка наличия флага – *set & flag == flag*



# Объекты.

C# – полностью объектно-ориентированный язык программирования. Здесь нет процедур или функций, которые бы не являлись членом какого-либо класса или его экземпляра. Причем все классы наследуются от System.Object. Кратко об основных элементах объектов (рис3):



В этом примере определены 2 класса – BaseExample и InheritExample. Подробнее, что обозначают данные определения:

* *public class* – объявляемый класс является публичным, т.е. его могут использовать за пределами модуля, в котором он собирается.
* *abstract class* – класс абстрактный. Он содержит абстрактные методы, поэтому нельзя создавать его экземпляры.
* *protected int m\_value* – член класса типа *int*, доступен только методам самого класса и пронаследованным от него.
* *public int Value* – свойство класса, представляет собой 2 функции, setter и getter. Для пользователя обращение к свойству синтаксически такое же, как к полю. Можно задавать различную видимость для setter’a и getter’a.
* *abstract void SomeMethod()* – абстрактный метод, т.е. не имеющий реализации в данном классе.
* *virtual void SomeAnotherMethod()* – виртуальный метод класса.
* *InheritExample : BaseExample, IDisposable* – второй класс наследуется от первого, а также от интерфейса *IDisposable*. Любой класс может наследоваться только от одного класса и любого количества интерфейсов.
* *public InheritExample(int value)* – публичный конструктор типа. Вызывается при создании экземпляра. Можно определить несколько конструкторов с различными параметрами. Конструктора не может быть у абстрактного класса.
* *~InheritExample()* – деструктор класса. Вызывается, когда сборщик мусора собирается удалить экземпляр класса. В отличие от других языков деструктор не удаляет из памяти объект и не может удалить другие, он только должен освободить занятые объектом ресурсы.
* *override void SomeMethod()* – переопределённый метод. Если переменная типа *BaseExample* указывает на объект типа *InheritExample* и вызывает метод *SomeMethod*, будет вызван метод класса *InheritExample.*
* *new void SomeAnotherMethod()* – переопределяет метод наследуемого класса с заменой. Если переменная типа *BaseExample* указывает на объект типа *InheritExample* и вызывает метод *SomeMethod*, будет вызван метод класса *BaseExample.*
* *public static InheritExample operator +(InheritExample a, InheritExample b)* – переопределяет оператор.
* *public void Dispose()* – реализация метода интерфейса *IDisposable.* Интерфейсы – просто набор абстрактных методов. Если класс – неабстрактный и наследует интерфейс, то он должен иметь реализации всех методов интерфейса.
* *static BaseExample GetExample()* – статичный метод класса. Не имеет доступа к не статичным членам класса, но сам может использоваться ими, а также вне класса, используя следующий синтаксис: *InheritExample.GetExample().*

# Reference-type и Value-type.

Существуют 2 вида типов - Reference-type и Value-type. Значения Reference-type находятся в управляемой куче, а переменная хранит только ссылку на данные. Таким образом, если одной переменной Reference-type присвоить другую переменную, то обе переменные будут указывать на одни и те же данные, то есть изменения в одной приводят к изменениям и в другой. Значения Value-type лежат в стеке потока, то есть данные в самой переменной. При присваивании происходит копирование объекта.

Создавая новый тип, нужно выбрать, какого вида будет этот тип. Рекомендуется, чтобы новый тип обладал следующими свойствами, чтобы его назначили как Value-type:

1. Тип должен быть достаточно примитивным, не должен выполнять что-то сложное.
2. Крайне рекомендуется, чтобы он был неизменным. То есть все его поля были readonly, заполнялись только в конструкторе, а любые изменения приводили бы к новой копии объекта.
3. Нет необходимости, чтобы тип наследовался от какого-то другого типа и чтобы какой-то другой тип наследовался от него.
4. Если предполагается, что тип может быть параметром функций, то общий размер типа не должен быть большим (рекомендуется менее 16 байт).

Чтобы создать Value-type, нужно объявить новый тип структурой (struct). Также Value-type являются энумераторы. Структуры напрямую наследуются от абстрактного класса System.ValueType, который наследуется от System.Object. Его главное отличие от System.Object в переопределенном методе Equals. Value-type не могут иметь потомков, поэтому отсутствуют виртуальные функции и прочее.

# Упаковка и распаковка.

При присваивании Value-type к переменной типа Reference-type, например, к переменной типа System.Object, происходит копирование объекта в управляемую кучу. Этот процесс называется boxing или упаковка. Обратный процесс – копирование из управляемой кучи при присваивании – unboxing или распаковка.



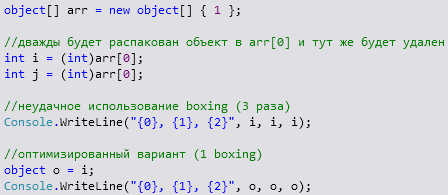
Эти операции не равносильны по производительности. Boxing более медленная операция, включает в себя следующие действия:

1. Выделяется память в управляемой куче под данные переменной, как при создании нового объекта Reference-type.
2. Копируются данные из стека в управляемую кучу.
3. Возвращается адрес созданного объекта.

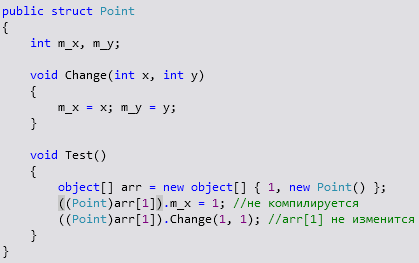
Unboxing включает в себя 2 стадии – получение адреса объекта в управляемой куче и копирование данных в переменную типа Value-type.

Таким образом, следует обращать внимание при работе с boxing/unboxing на следующее:

1. По возможности избегать излишней упаковки/распаковки одного и того же объекта.



1. Нельзя изменять значения в только что распакованных объектах (без присваивания переменной).



1. Не рекомендуется изменять значения упакованных объектов, используя интерфейсы.

