#### Платформа .NET

**Платформа .Net Framework** представляет собой совокупность программных модулей в рамках которых в ОС Windows выполняются **.Net** приложения  
 **.Net Framework** - это инсталляционный пакет который автоматически устанавливается при установке ОС Windows, также есть в свободном доступе на сайте Microsoft

Основой платформы является общеязыковая среда исполнения **Common Language Runtime** (CLR).

Платформа **Microsoft .NET** — это **технология, которая поддерживает создание и выполнение приложений самых разных типов (консоль, рабочий стол, веб, мобильные, микросервисы, облако, ML, разработка игр, Internet of Things**).

Главной идеей разработки .NET было стремление сделать кроссплатформенную виртуальную машину для выполнения одного и того же кода в различных ОС.

Одной из целей разработки новой платформы было объединение всех наиболее удачных наработок в рамках единой платформы и их унификация. Кроме того, ставилась задача следования всем актуальным тенденциям в области программирования на тот момент. Так, например, новая платформа должна была напрямую поддерживать объектно-ориентированность, безопасность типов, сборку мусора и структурную обработку исключений.

Вероятно, самым большим достижением .NET остаётся **межъязыковое взаимодействие** (language interoperability). Впервые в истории программирования появляется единая модель, позволяющая на равных пользоваться различными языками для создания приложений. Так как MSIL не зависит от исходного языка программирования или от целевой платформы, в рамках .NET становится возможным развивать новые программы на базе старых программ.

#### Процесс компиляции и запуск приложения

#### CLR

**CLR** - общеязыковая среда исполнения

1. Отвечает за компиляцию и выполнение **.Net** программ.

2. Управляет ресурсами.

3. Генерирует нужное исключение.

Компоненты CLR

1. **Сборка** - формируется после компиляции состоит из 4ех компонентов

- **Метаданные** - инфа о типах

- **Манифест** - тех инфа о сборке

- **Байт-код** (высокоуровневый ассемблер) в такой вид переводится откомпилированная программа в среде **.Net**, в отличие от низкоуровнего exe файла на уровне байт кода существует понятие объекта

- **Ресурсы** - это доп файлы (картинки, звук), которые подключаться извне

2. **VES** (Virtual Execution System или Виртуальные процессор) - центральные понятие **CLR**. Единицей компиляцией для **VES** является сборка.

3. **JIT** компиляция (Just In Time) запускается **VES** процессором, компилирует методы которые вызваны с дальнейшем кешированием, после чего создается exe файл и отправляется к реальному процессору.

- **Кэширование**. При **кэшировании** память выделяется только при первом вызове метода и не уничтожается после использования, при последующих вызовах будет использоваться та же функция. Таким образом первое выполнение **.Net** программ медленее происходит чем последующие вызовы.

#### 

#### **Компоненты платформы .Net Framework**

1. **FSL** - все классы .Net

2. **BSL** - подмножество **FCL**. Необходимый минимум для всех **.Net** совместимых языков и технологий

3. **CTS** - описание всех типов

4. **CLS** - это минимальные требование к языкам программирования для поддержки **.Net**

#### **BCL**

Коллекции в BCL

**Коллекция** - это класс, который размещает совокупность каких либо данных, объектов в оперативной памяти по спец принципу.

**Коллекции** в BCL бывают двух категорий

**1.** **Обобщенные**

**Проблема**: при использовании **обобщенных** коллекций на упаковку и распаковку отводится много времени. Для решения этой проблемы были созданы **обобщенные** коллекции, в которых тип определяется на этапе компиляции программы.

a. **List**<T> - массив

b. **LinkedList**<T> - двусвязный список (односвязного нет в BCL

c. **Stack**<T> - массив

d. **Queue**<T> - массив

e. **PriorityQueue**<T> - массив

f. **Dictionary**<T> - массив

g. **SortedList**<T> - сортированный словарь с массивом

h. **SortedDictionary** - словарь с деревом

**2.** **Необобщенные.**

Все элементы это object.

a. **ArrayList** - динамическая структура (массив)

b. **Stack** - идеология LIFO (массив)

c. **Queue** - идеология FIFO (массив)

d. **SortedList** - сортировка по ключам (массив). Содержит два массива (ключей и значений). Так как сравнение идет по ключам, ключи должны быть однотипные.

e. **HashTable** - словарь (key, value), внутри массив. Представляет собой совокупность объектов **DictionaryEntry**. У каждого объекта есть ключ значение.

#### IL код

IL код (Intermediate Language) или CIL код (Common Intermediate Language) — это промежуточный язык, используемый в платформе .NET. IL код играет ключевую роль в процессе выполнения программ, написанных на языках .NET, таких как C#, VB.NET, и F#.

**Что такое IL код?**

- Промежуточный язык: IL код представляет собой низкоуровневый язык, который находится между исходным кодом на языке программирования и машинным кодом, который исполняет процессор. Он служит промежуточной стадией в процессе компиляции и выполнения программ.

- Платформенно-независимый: IL код не зависит от конкретной платформы или архитектуры процессора. Это позволяет программам, написанным на .NET языках, быть совместимыми с разными платформами.

Как работает IL код?

1. Компиляция:

- При компиляции исходного кода на C# или другом .NET языке, код сначала компилируется в IL код, а не в машинный код. Например, компилятор C# (csc) преобразует C# код в IL код и сохраняет его в файл с расширением `.dll` или `.exe`.

2. JIT-компиляция:

- Когда приложение запускается, .NET среда выполнения (CLR — Common Language Runtime) компилирует IL код в машинный код, подходящий для конкретной архитектуры процессора, используя Just-In-Time (JIT) компиляцию. Этот процесс происходит во время выполнения программы.

3. Управляемая среда:

- IL код исполняется в управляемой среде .NET, что обеспечивает безопасность, сборку мусора и другие услуги. Это позволяет коду работать корректно и безопасно на различных платформах.

Основные характеристики IL кода:

1. Платформенная независимость: IL код не привязан к конкретной аппаратной платформе.

2. Интероперабельность: Разные языки .NET могут взаимодействовать друг с другом, потому что все они компилируются в один и тот же IL код.

3. Управляемое выполнение: IL код выполняется в управляемой среде, что обеспечивает безопасность и управляемое управление памятью.

IL код представляет собой промежуточный шаг в процессе компиляции и выполнения программ на платформе .NET. Он обеспечивает платформенную независимость и позволяет программам на разных языках .NET быть совместимыми и работать в управляемой среде выполнения.

#### **Типы данных**

Все типы в **.Net** (даже те которые мы сами создаем и не реализуем явное наследование) наследуются от класса System.Object (Наследование).

Все типы в .Net делятся на две категории:

**1.** **Ссылочные**

**2.** **Значимые**

| **Ссылочные** | **Значение** |
| --- | --- |
| class | struct, enum |
| Способ передачи в метод по ссылке | Способ передачи по значению |
| Размещение в куче | Размещение в стеке если не являются полем класса |
| Наследуются | Не наследуются |
| Конструктор по умолчанию переопределяется | Конструктор по умолчанию не переопределяется |

Все примитивные типы (int, double) в **.Net** являются структурами.

**Алиас** - это сокращенное название для значимых типов. Например для структуры Int32 **алиасом** является int.

Конструктор по умолчанию для значимых типов вызывается явно с помощью оператора **new**. При этом к выделению памяти в куче **new** не имеет никакого отношение

int a;

Console.Write(a); // Ошибка потому что конструктор по умолчанию не вызывается.

int a = new int();// Выхывается констурктор по умолчание.

Console.Write(a); // Будет 0.

Для ввода с клавиатуры используется метод **Console.ReadLine()**, который возвращает только строку.  
 При необходимости мы используем класс **Convert** для получения желаемого типа. Или метод **Parse**, которые есть у любой структуры.

#### **Массивы**

Все **массивы** - классы, наследуются от одного базового класса **SystemArray**. Так как **массив** это ссылочный тип данных, память выделяется в куче. Ссылочный **массив** имеет свои преимущества и недостатки

1. Преимущество - при наследовании **массив** получает необходимую функциональность от базового класса Array.

2. От того, что **массивы** размещаются в куче замедляется доступ к ним.

Типы массивов в .Net

**1.** **Одномерные**

**2.** **Многомерные**

3. **Зубчатые** - разновидность многомерных **массивов**, каждая строка которой может содержать разное количество элементов. Экономит память, но уменьшается скорость обращения.

Основные классы метода Array

1. **IndexOf** - поиск первого вхождения числа в массиве

2. **LastIndexOf** - поиск последнего вхождения числа в массиве

3. **Sort** - сортировка по возрастанию, реализация по быстрой сортировки

4. **Reverse** - переворачивает массив

5. **Copy** - копирует один массив в другой, параметры (откуда, куда, сколько)

6. **Clear** - обнуление массива

7. **GetLength** - Количество строк двумерного массива

#### **Строки(особенности в .NET)**

**Строка** - это ссылочный тип данных. Строки в **.Net** представляют собой набор **unicode** символов. Класс **System.String** позволяет создавать не изменяемые строки.

#### **Преобразование типов**

**Упаковка** - это неявное преобразование любого значимого типа в object.

При **упаковке** в куче выделяется место для object и записывается туда само значение справа от равно ( = ).

**Распаковка** - это неявное преобразование object в значимый тип.

При **распаковке** в стеке выделяется место для значимого типа и значение переносится из кучи в стэк.

**Упаковка и распаковка** очень сильно бьет по производительности.

#### **Делегаты**

**Делегат** - это ссылочный тип, который представляет собой указатель на метод.

**Пространство** System.Delegate и System.MulticastDelegate.

**Синтаксис**

**Делегат** (тип делегата) можно объявить за пределами класса или структуры.

Метод **Invoke()** вызывает метод адрес который содержиться в **делегате**.

Каждый объект **делегата** содержит в себе массив адресов, с помощью операции **+=** адрес метода добавляется в массив, **-=** адрес удаляется из массива.

Для получение инкапсулированного массива адресов используется метод **GetInvocationList()**.

**Мультикастинг** - представляет собой цепочку вызовов с помощью одного объекта **делегата**.

#### Интерфейсы

**Интерфейсы** в **.Net** относятся к ссылочным типам.

**Интерфейсы** создаются для реализации в разных иерархиях вне зависимости от типа.

**Интерфейсные** ссылки более абстрактные сущности чем ссылки базового абстрактного класса.

С точки зрения архитектуры классов - **интерфейсы** не должны содержать поля.

**Интерфейсы** не наследуются, а реализуются.

Класс может наследовать данные только от одного родителя и при этом может реализовать несколько интерфейсов.

Интерфейсные свойства

Помимо прототипов **интерфейсы** могут содержать в себе автоматические свойства и свойства для индексатора.

**Интерфейс** - это контракт (набор требований) для будущих классов, которые будут его реализовать. Условия необходимо соблюдать.

Абстрактный класс

1. **Абстрактный класс** не может иметь собственных объектов. Является базовой минимальной основой для своих потомков.

2. **Абстрактный класс** - это заготовка для своих производных классов.

3. **Абстрактный класс** имеет минимальное базовое поведение (поля, реализованные методы).

**Стандартные интерфейсы**

**.Net** представляет набор стандартных **интерфейсов** для решения часто возникающих проблем.

**IComparer**. Интерфейс содержат метод compare реализация которого в классе позволяет сравнивать между собой переданные компоненты.

**IComparable**. Содержит 1 метод **CompareTo.**

**ICloneable**.Выполняет глубокое копирование, создает клон объекта.

#### **Перегрузка операции**

Перегрузка операций в **.Net** осуществляется с помощью открытых статических методов класса (public, static)

Не перегружаемые операторы

- [], ()

- new

- is, as

- sizeof, typeof

- тернарные операции

- оператор присваивания

При перегрузке **бинарных** операций статический метод принимает 2 операнда (Потому что нет указателя this.).

При перегрузке **унарных** операций 1 операнд.

Перегрузка логических операций

Логические операции перегружаются только парами

- < и >

- == и !=

- <= и >=

**Перегрузка операторов преобразований**

**Explicit** позволяет реализовать явное преобразование

**Implicit** позволяет реализовать неявное преобразование

Внутри класса нельзя одновременно реализовать и **explicit**, и **implicit** преобразование с одинаковыми сигнатурами.

**Implicit** срабатывает при явно и при неявном преобразовании.

#### **Наследование**

При делегировании между конструкторами используется ключевое слово **this**.

**.Net** поддерживает только одиночное наследование.

Для вызова методов базового класса и конструктора используется слово **base**.

**Абстрактный** метод обязательно необходимо переопределить в потомках.

Объявляется с помощью слова **abstract** и не имеет собственного поведения в базовом классе.

Класс можно сделать **абстрактным**. **Абстрактный** метод можно создать только в абстрактном классе.

#### **События**

**Событие** - это объект, который инициирует определенное действие и на это действие реагируют подписчики (наблюдатели).

Для создание объекта **события** (event) обязательно нужен **делегат**.

**Синтаксис**

[модификатор доступа] event ИмяДелегата ИмяСобытия.

С помощью **делегата** определяется сигнатура подписчиков.

**События**, как и **делегаты**, содержат в себе массив наблюдателей (Invocation List).

В большинстве случаев **события** можно заменить **делегатами**. Но с технической точки зрения, есть отличия между ними:

- Объект **делегата** может существовать без другого объекта

- **События** не может существовать без объекта инициатора.

Список наблюдателей **событий** контролируется с помощью двух операция  
 **+=** и **-=** (**=** запрещено)

Есть понятие **“Событийное свойство”**, они позволяют контролировать доступ к объекту **event** если тот инкапсулирован в классе. Также благодаря **событийным свойствам** можно явно создать список наблюдателей и реализовать доступ к этому списку.

**Событийное свойство** представляет собой конструкцию, которая создается по следующему синтаксису

#### **Файловые потоки**

class **StreamReader** и **StreamWriter**. Данные классы по умолчанию работают с текстовым форматом (текст файлы).  
 **StreamReader** - чтение файлов  
 **StreamWriter** - запись в файл

Метод **Peek()** - возвращает следующий доступный символ или -1 если нет символа

**FileStream** - байтовый класс. Обеспечивает доступ на уровне байт

class **BinaryReader** и **BinaryWriter** - это классы обертки для работы в бинарном формате.

**BinaryReader** содержит ряд методов для чтения данных определенного типа.

**Работа с директориями и папками**

Для работы с директориями используются классы **Directory** и **DirectoryInfo**

Для работ с файлами используются классы **File** и **FileInfo**

Классы **Directory** и **File** - статические

#### **Сбор мусора**

**Управляемая куча** представляет собой набор поколений, их 3 (0, 1, 2).

**0 поколение** имеет минимальный размер среди всех поколений. Все объекты ссылочных типов сначала попадают в **0 поколение**.

Когда **0 поколение** заполняется и в коде создается новый объект ссылочного типа впервые запускается алгоритм сборки мусора.

**Алгоритм сборщика мусора:**

**1.** **Отслеживание ссылок**

a. Проверяет на каком участке есть активные ссылки

**2.** **Маркировка.**

a. Участки, которые имеют активные ссылки в стеке маркируются

b. После проверки всех объектов, куча содержит набор маркированных и немаркированных объектов

c. Маркированные объекты переживут уборку мусора потому что на них ссылается хотя бы 1 объект в стеке. Немаркированные недостижимы потому что в приложении не существует корня через которое приложение могло бы к ним обратиться, в стеке нет указателя

**3.** **Сжатие**

a. После сжатия все достижимые участки располагаются строго последовательно в текущем поколении.

**4.** **Перенос**

a. Перенос достижимых объектов в следующее поколение

**5.** **Выделение памяти**

a. Выделение памяти для нового объекта в 0 поколении

**Вопрос?** Сколько объектов в 0 поколении после первого запуска сборщика мусора?

**Ответ**: В 0 поколении только новый объект. Все старые достижимые объекты перешли в следующее поколение

**Предположения**

1. Чем младше объект тем короче его время жизни

2. чем старше объект тем длиннее его время жизни

3. уборка мусора в части кучи выполняется быстрее чем во всей куче

**Алгоритм сборщика мусора** корректно работает только с простыми объектами. Объекты которые содержат в себе ссылки на файлы или БД не будут корректно уничтожены из управляемой кучи.

Для корректного удаления таких объектов **.Net** предоставляет несколько способов:

1. Интерфейс IDisposable. Есть метод Dispose().  
 При использовании using для выделенных объектов, автоматически будет вызываться метод DIspose() до того как освободиться место в управляемой куче

2. Использование финализатора. При создании “деструктора” в классе происходит не явное переопределение метода Finalize() для текущего класса. В блок try этого метода попадает тело “деструктора”

При использовании **using** Dispose() вызывается автоматически при выходе из области **using**

class System.GC позволяет явно вызвать сборку мусора в указанном поколении

#### **Сериализация объектов**

Существует возможность добавления дополнительных сведений типах в метаданные, для этого используются **атрибуты**.

Пространство **System.Attribute**

Для создания собственного атрибута необходимо наследоваться от базового класса **System.Attribute**

Удобство **атрибутов** заключается в том, что можно внести инфу в сборку, а другой разработчик может достать эту инфу, и в зависимости от полученных данных менять поведения логику работы с классом.

Метод **GetType()** возвращает тип объекта.

Метод класса Type - **GetCustomAttribute()** возвращает список атрибутов для конкретного типа в виде массива Object.

**Сериализация** - процесс преобразования какого либо объекта в поток байтов.

**Десериализация** - процесс преобразования из байтов в объекта.

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

#### 

#### 