Разработка приложений на платформе .NET

Лекция 13

Время жизни объектов

MSIL-компиляция

Загрузка CLR

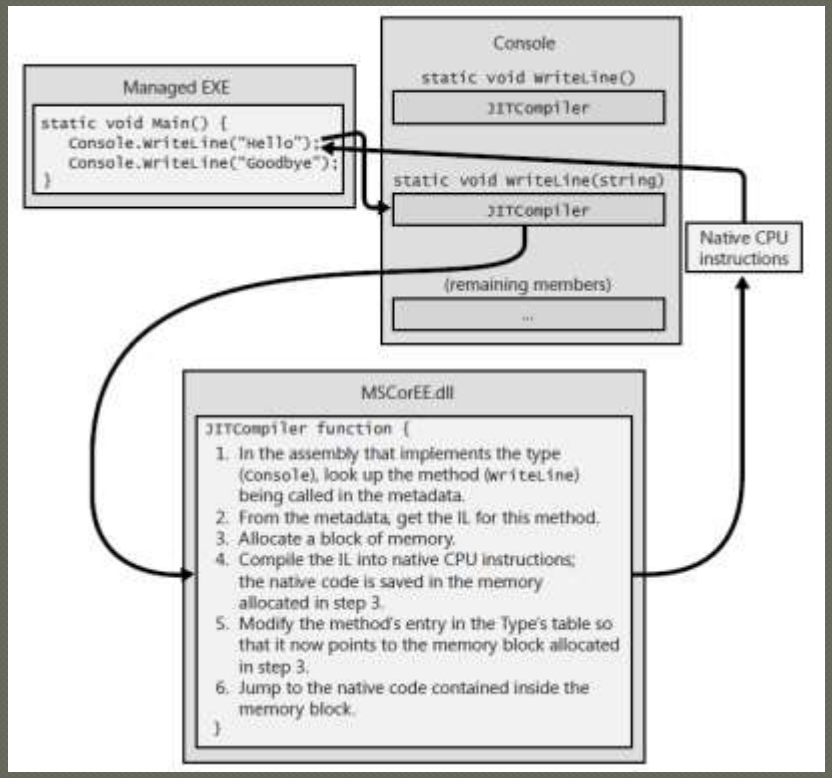
- При запуске **exe** файла **Windows** анализирует заголовок **exe** файла для определения разрядности адресного пространства 32 или 64 бит (**PE32** или **PE32+**)
- В адресное пространство процесса **Windows** загружает соответствующую версию **MSCorEE.dll** (**x86**, **x64**, **ARM**)
- Основной поток вызывает метод в **MSCorEE.dll**, инициализирующий **CLR**, загружающий сборку **exe** и вызывающий метод **Main** сборки.

Компиляция	Заголовок	x86 Windows	x64 Windows	ARM Windows RT
AnyCPU	PE32 /независимый	Выполняется как 32-bit приложение	Выполняется как 64-bit приложение	Выполняется как 32-bit приложение
AnyCPU Prefer 32 bit	PE32 /независимый	Выполняется как 32-bit приложение	Выполняется как WoW64 приложение	Выполняется как 32-bit приложение
x86	PE32 /x86	Выполняется как 32-bit приложение	Выполняется как WoW64 приложение	Не выполняется
x64	PE32+ /x64	Не выполняется	Выполняется как 64-bit приложение	Не выполняется
ARM	PE32+ /Itanium	Не выполняется	Не выполняется	Выполняется как 32-bit приложение

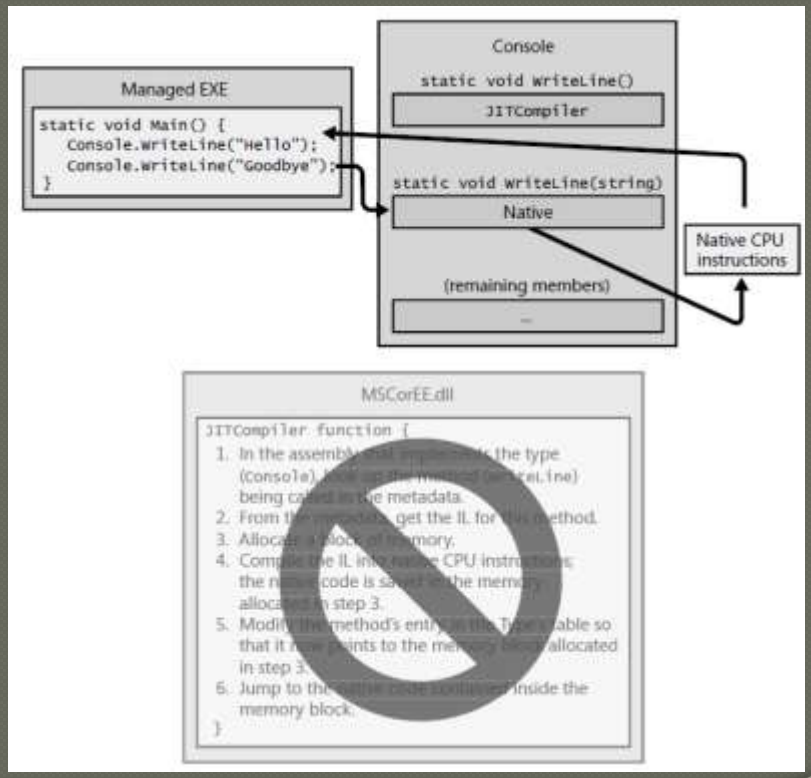
Исполнение кода

- Перед вызовом **Main**
  - Находятся все типы, использованные в **Main**.
  - Создаются внутренние структуры для каждого типа, содержащие записи для каждого метода. Каждая запись содержит адрес, с реализацией метода.
  - При инициализации в каждую запись устанавливается адрес спец. функции из **MSCorEE – JITCompiler**

При первом вызове функции (**WriteLine**) в методе **Main**



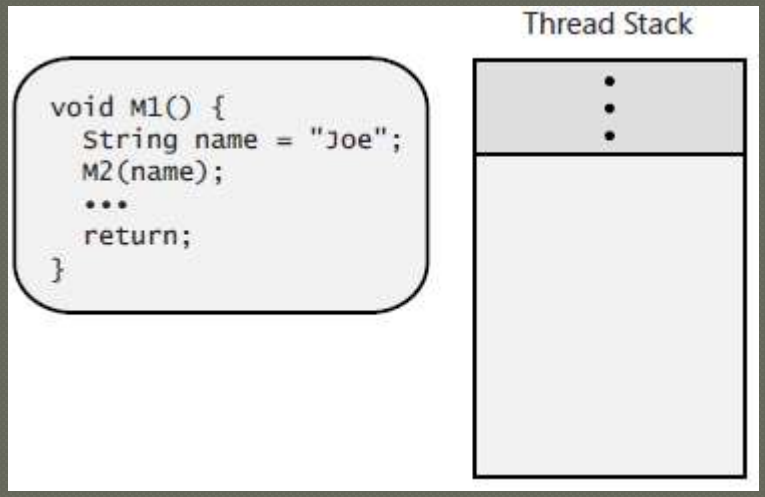
При последующих вызовах функции (**WriteLine**) в методе **Main**



См. Jeffrey Richter "CLR via C#"

Выполнение кода

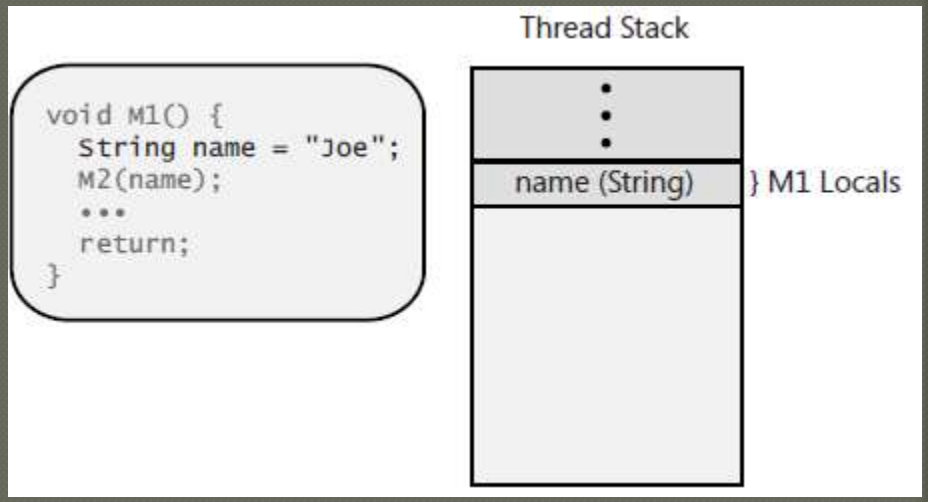
- При создании потоку выделяется стек в 1Мб
- Стек используется для передачи параметров в методы и для хранения локальных переменных
- Каждый метод содержит входной код, инициализирующий метод и выходной код, выполняющий очистку и возвращающий управление вызывающему коду



См. Jeffrey Richter "CLR via C#"

Выполнение кода

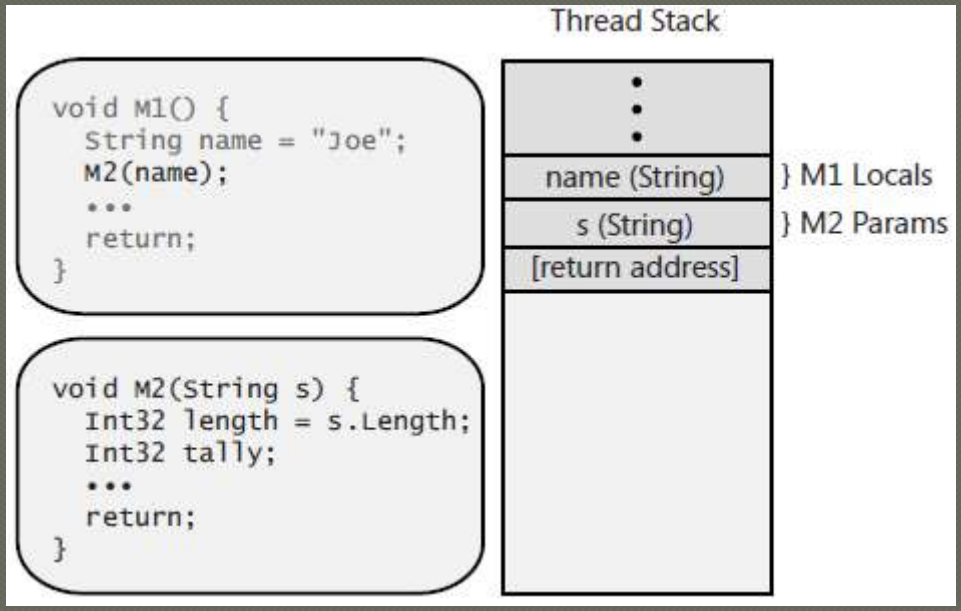
- В стеке выделяется память под локальную переменную **name**



См. Jeffrey Richter "CLR via C#"

Выполнение кода

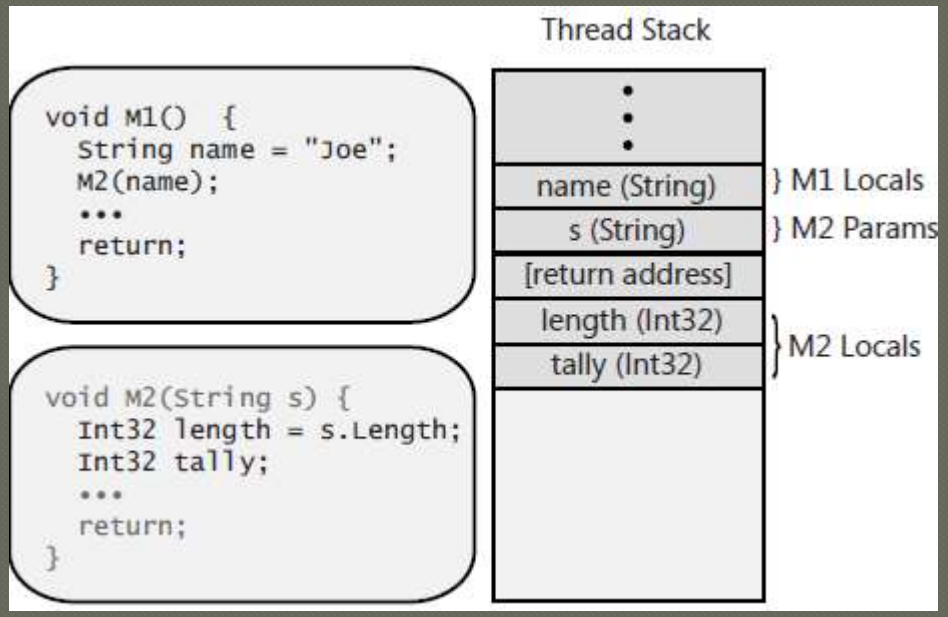
- Переменная **name** для передачи в метод **M2** заталкивается в стек
- В стек заталкивается адрес возврата. Адрес команды, следующей за вызовом метода **M2**



См. Jeffrey Richter "CLR via C#"

Выполнение кода

- В начале метода M2 в стеке выделяется память под локальные переменные
- При возвращении из методов, часть стека очищается



См. Jeffrey Richter "CLR via C#"

Выполнение кода в CLR

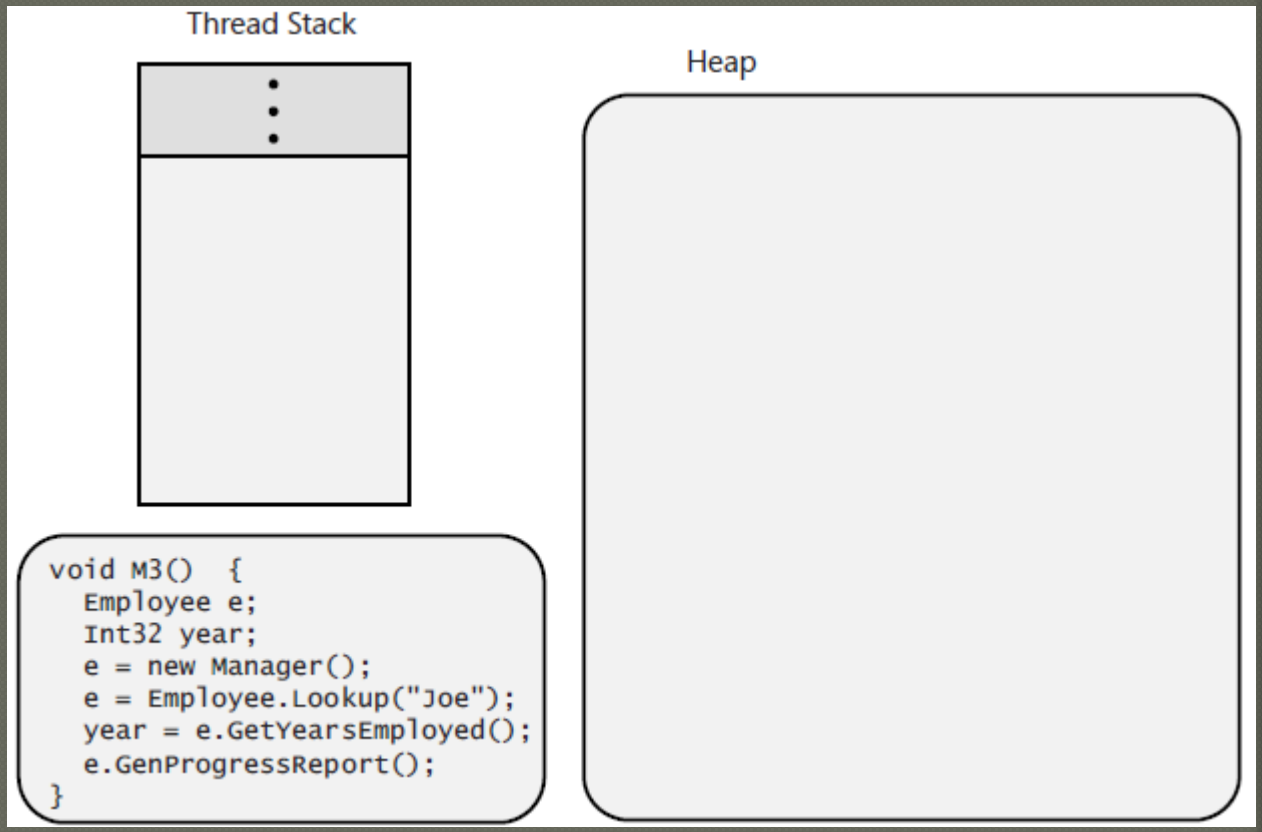
```
internal class Employee
{
        public Int32 GetYearsEmployed() { … }
        public virtual String GetProgressReport() { … }
        public static Employee Lookup(String name) { … }
}

internal sealed class Manager : Employee
{
        public override String GetProgressReport() { … }
}

void M3()
{
        Employee e;
        Int32 year;
        e = new Manager();
        e = Employee.Lookup("Joe");
        year = e.GetYearsEmployed();
        e.GetProgressReport();
}
```

Выполнение кода в CLR

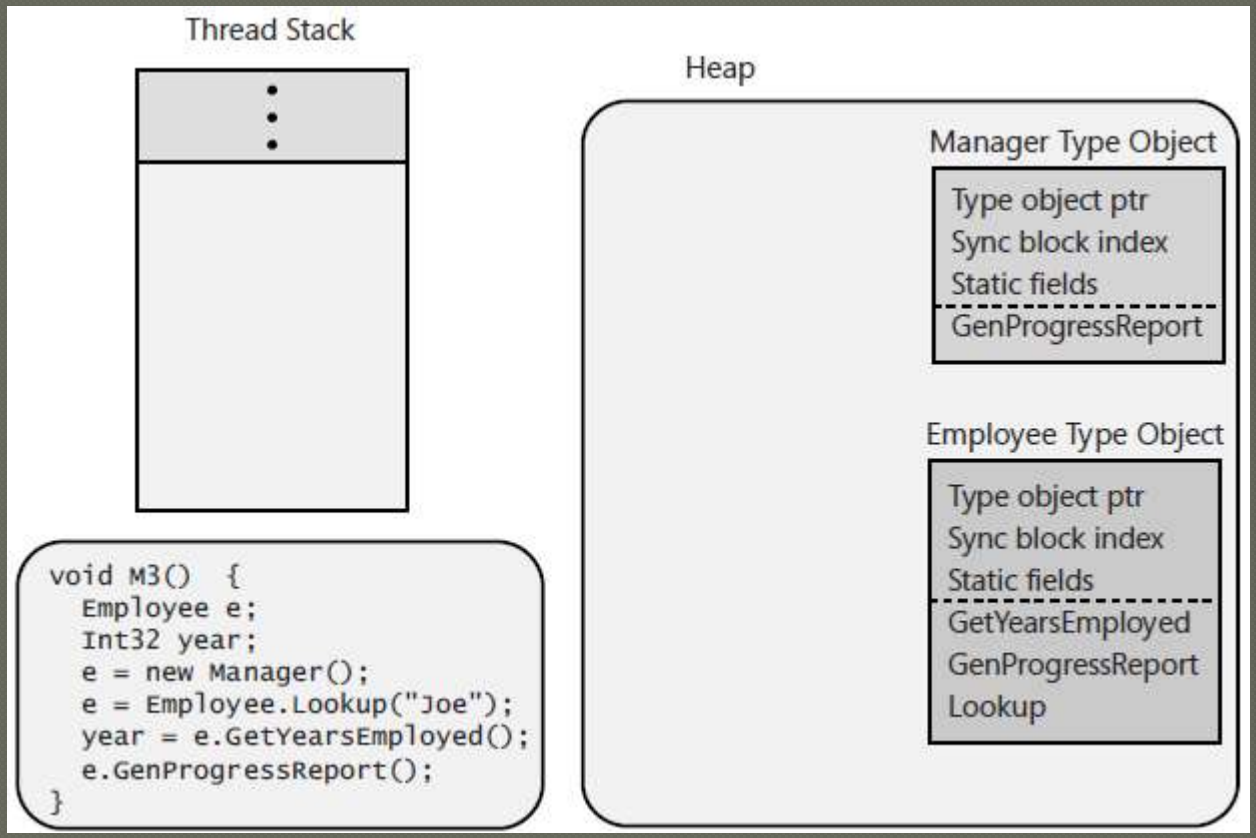
- Есть стек
- Для простоты пустая куча



См. Jeffrey Richter "CLR via C#"

Выполнение кода в CLR

- При компиляции метода M3 JIT-компилятор выявляет все типы, использующиеся в M3. Это Employee, Int32, Manager, String
- CLR загружает сборки, в которых содержаться нужные типы
- Используя метаданные сборок CLR получает информацию о типах и создает структуры данных для этих типов – объекты-типы
- Объекты типы содержат:
  - Указатель на объект-тип
  - Индекс блока синхронизации
  - Статические переменные типа
  - Таблицу методов
  - Указатель на объект-тип базового типа (не показан)

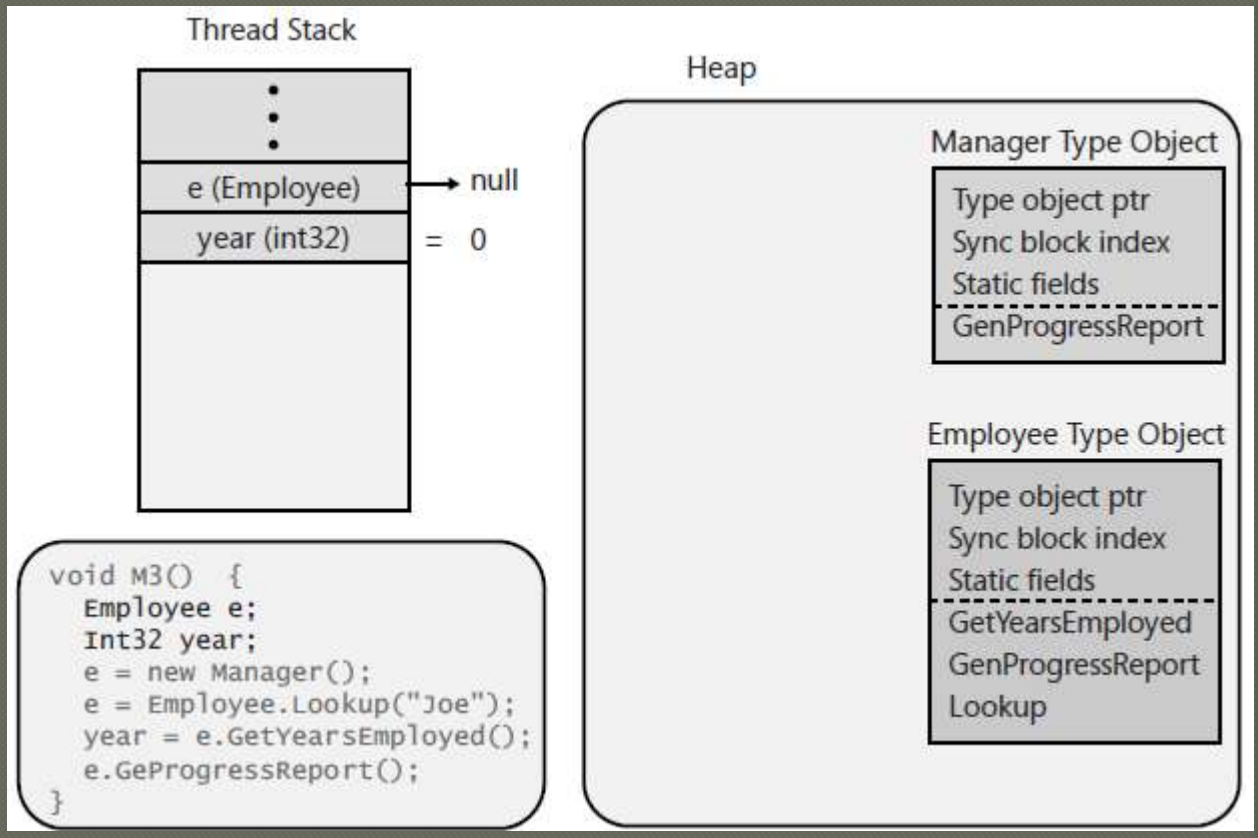


int и string для простоты не указаны

См. Jeffrey Richter "CLR via C#"

Выполнение кода в CLR

- При выполнении M3, в стеке потока выделяется память под локальные переменные.
- CLR автоматически инициализирует эти переменные значениями 0 или null (в рамках входного кода метода)



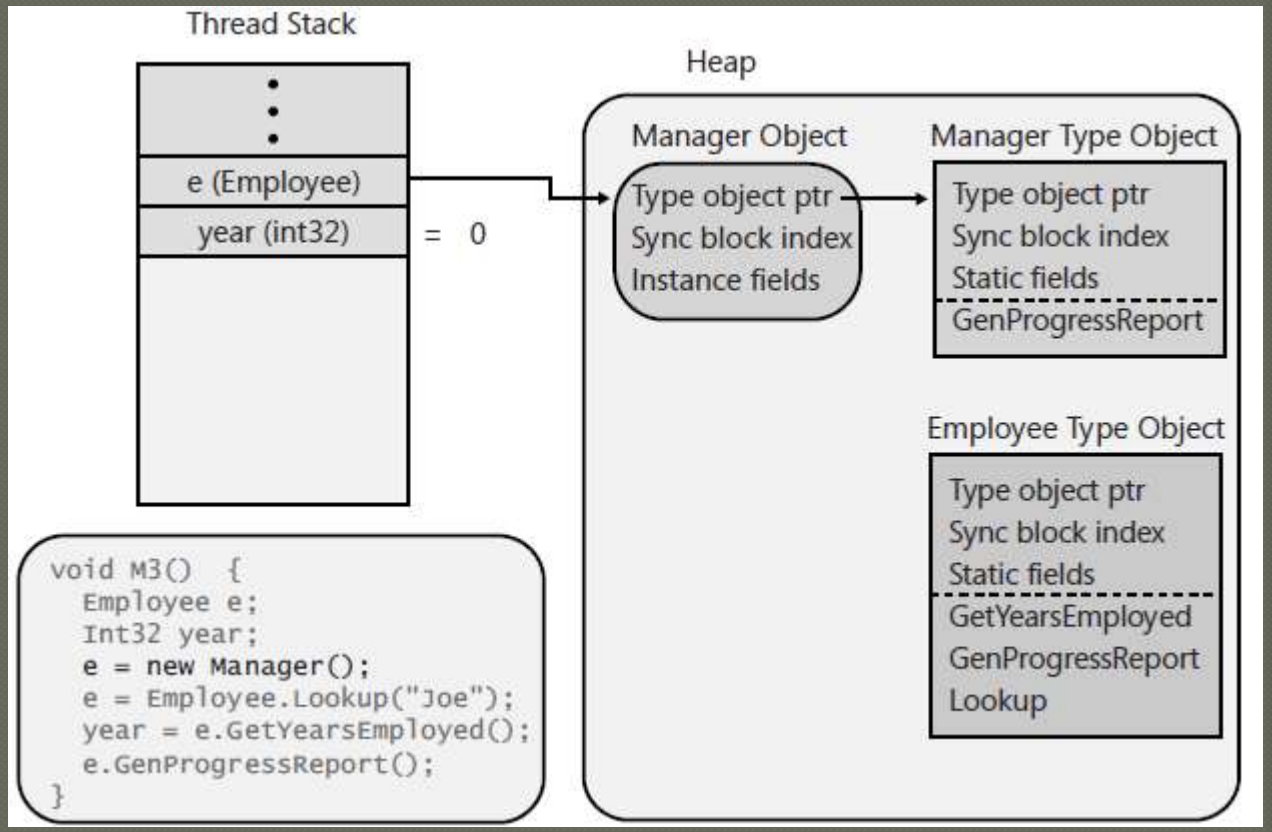
См. Jeffrey Richter "CLR via C#"

Действия оператора new

- Вычисление количества байт, необходимых для хранения всех экземплярных полей типа и экземплярных полей всех его базовых типов. Каждый объект в куче также содержит указатель на объект-тип и индекс блока синхронизации
- В куче выделяется память для объекта.
- Выделенные в куче байты инициализируются 0.
- Инициализируются указатель на объект-тип и индекс блока синхронизации
- Вызывается конструктор указанный при вызове **new.** При этом по цепочке сначала вызываются конструкторы базовых типов.
- Возвращается указатель на созданный объект

Выполнение кода в CLR

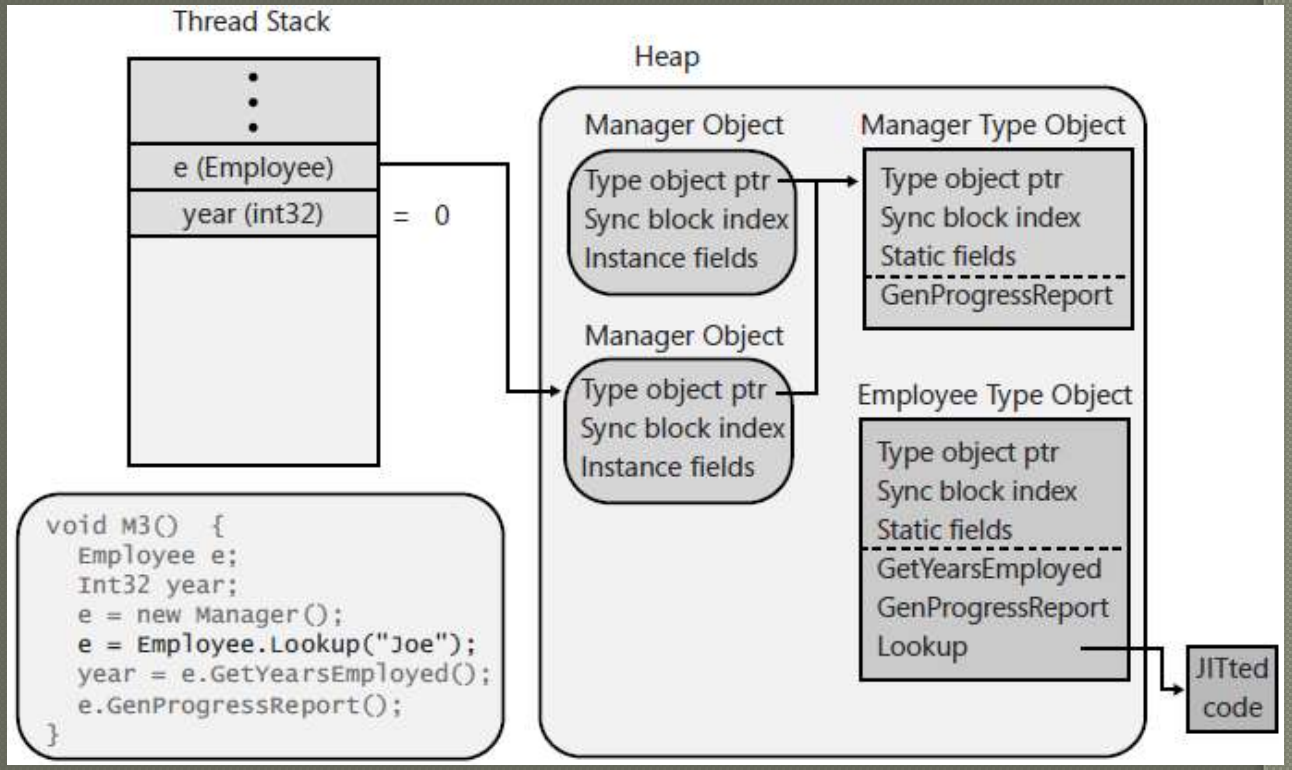
- Создается объект – экземпляр типа **Manager**
- При этом создаются и инициализируются:
  - Ссылка на объект-тип **Manager**
  - Индекс блока синхронизации
  - Экземплярных полей у **Manager** и его предков нет
- Возвращается указатель на созданный объект, который помещается в переменную **e** в стеке



См. Jeffrey Richter "CLR via C#"

Вызов статического метода

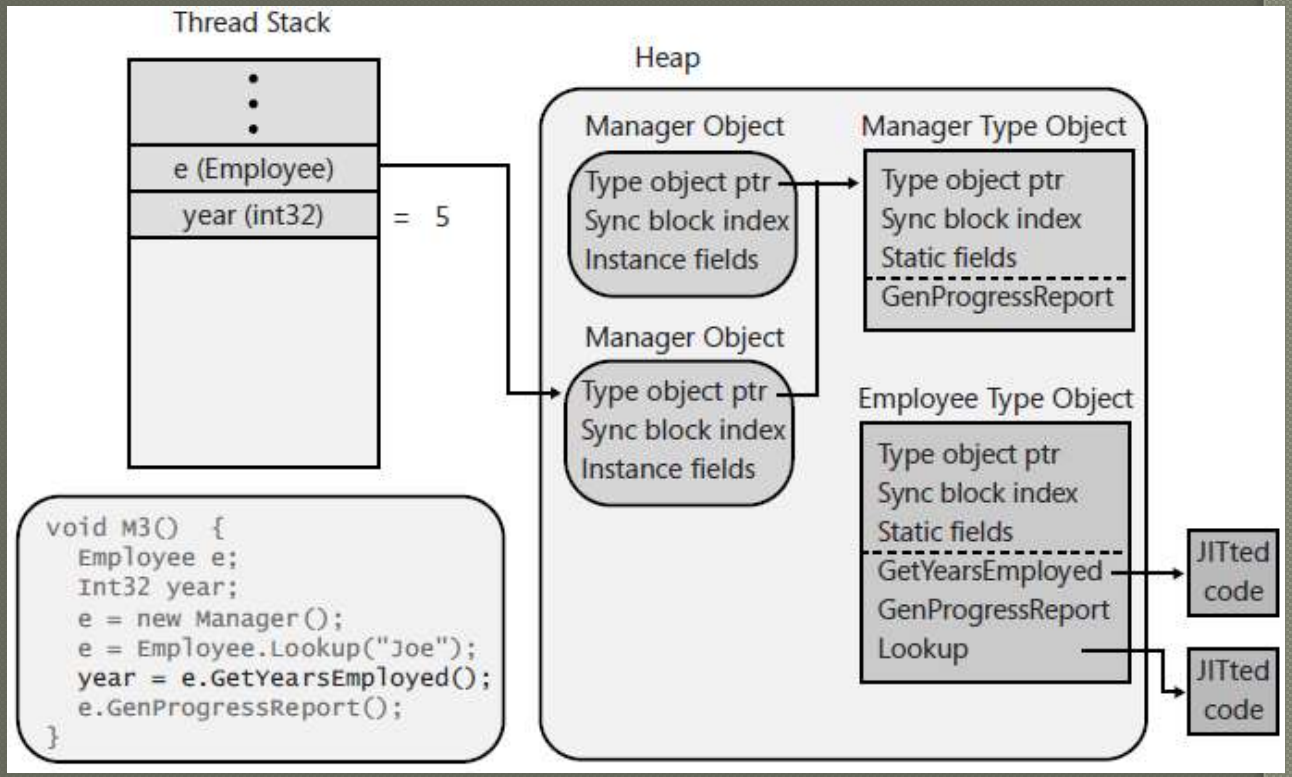
- CLR при вызове статического метода Lookup определяет местонахождение объекта-типа.
- На основе таблицы методов объекта-типа находит точку входа метода, при необходимости компилирует JIT компилятором и передает управление машинному коду.
- Предположим, что метод Lookup создает и возвращает новый объект Manager
- Адрес возвращенного объекта помещается в переменную e.



См. Jeffrey Richter "CLR via C#"

Вызов не виртуального метода

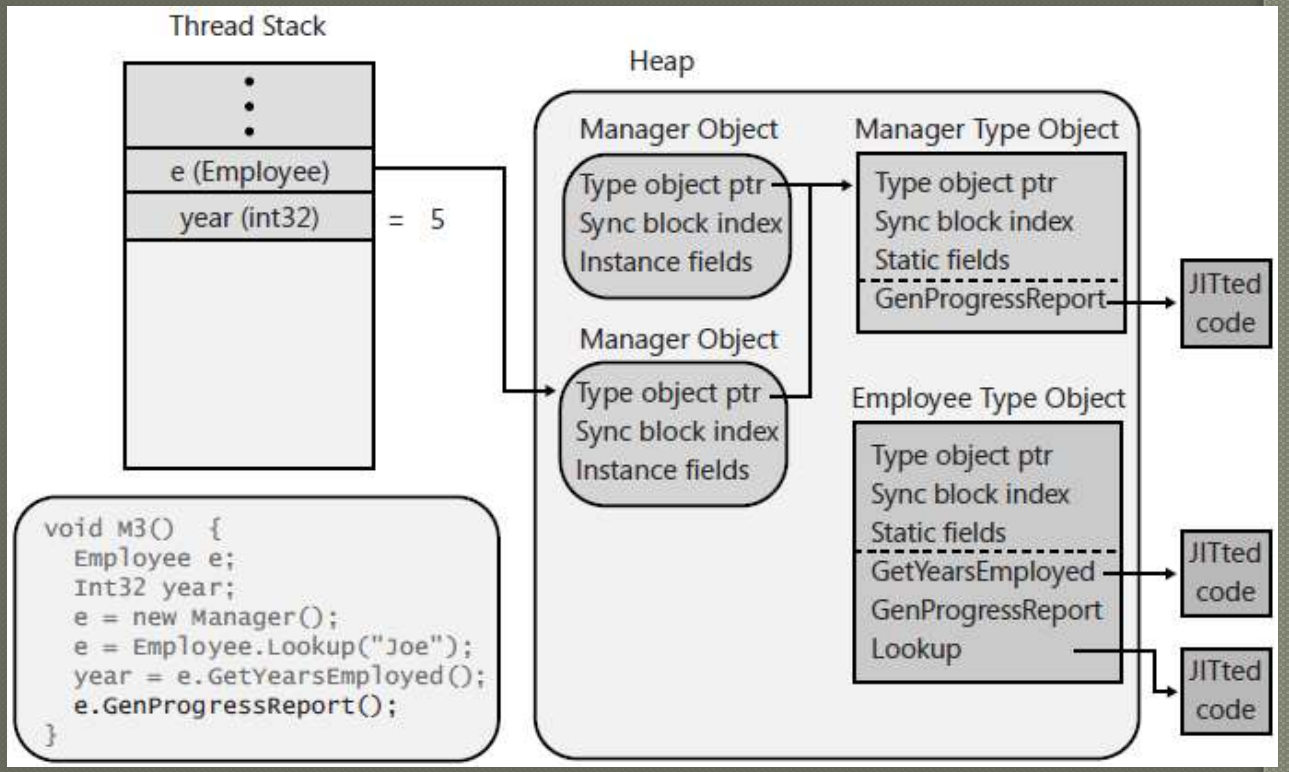
- При вызове не виртуального метода **GetYearsEmployed** CLR определяет местонахождение объекта-типа переменной
- Если объект тип не содержит определение вызываемого метода JIT-компилятор ищет метод по цепочке у объекта-типа предков
- При необходимости метод компилируется JIT компилятором и управление передается машинному коду.
- Предположим, что метод **GetYearsEmployed** возвращает 5. Int сохраняется в переменной **year**



См. Jeffrey Richter "CLR via C#"

Вызов виртуального метода

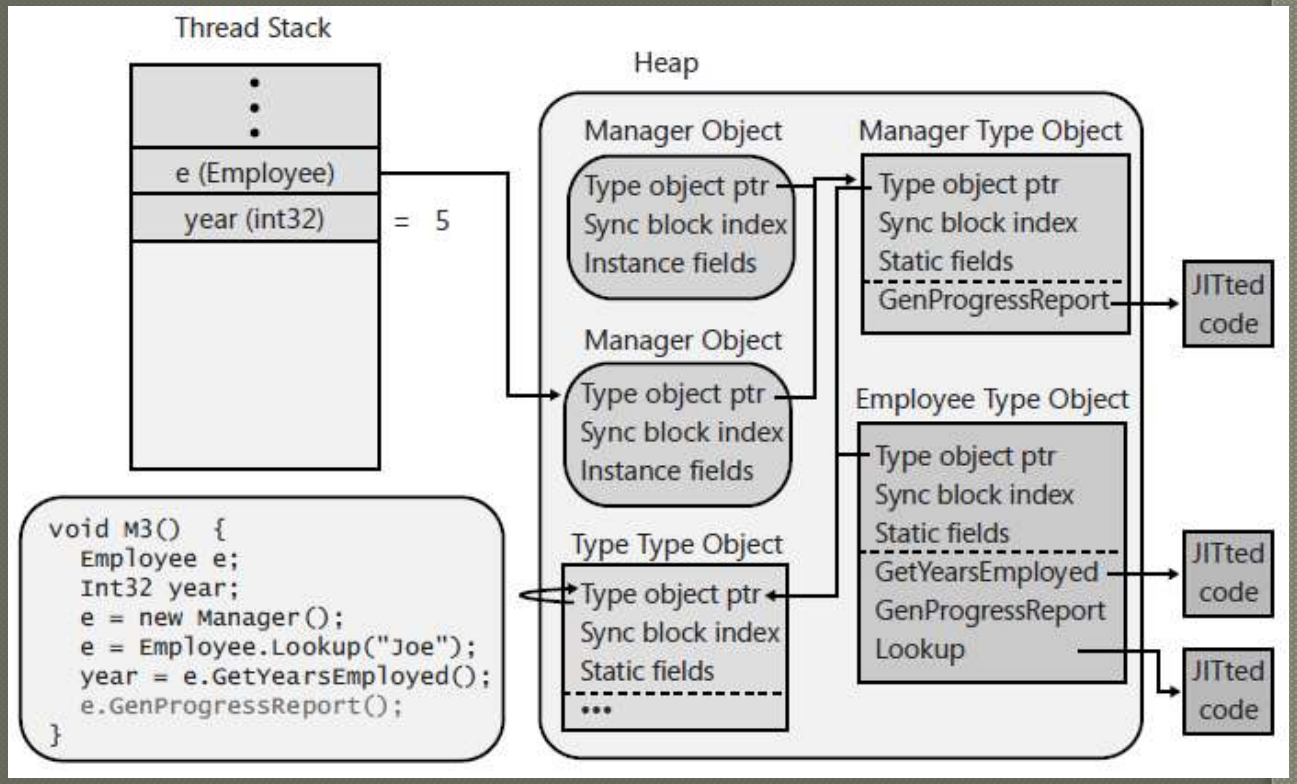
- Каждый раз при вызове виртуального метода (**GetProgressReport**) производится дополнительная работа. Вначале смотрится переменная (**e**), используемая для вызова. Затем смотрится реальный объект в переменной (объект **Manager**). Находится его объект-тип **Manager**. В таблице методов объекта-типа находится запись вызываемого метода
- При необходимости метод компилируется **JIT** компилятором и управление передается машинному коду.



См. Jeffrey Richter "CLR via C#"

Объекты-типы

- Объекты-типы тоже объекты
- Объекты-типы – это экземпляры типа **Type**. Поэтому их указатель на объект-тип ссылается она объект-тип **Type**
- Сам объект –тип **Type** своим указателем на объект-тип ссылается на самого себя



См. Jeffrey Richter "CLR via C#"

Время жизни объектов

- Когда объект становится ненужным, он становится кандидатом на удаление сборщиком мусора
  - Например:
    - При выходе из область видимости переменной
    - При выходе из метода – все локальные переменные, которые не возвращаются методом и не используются в параметрах **ref** и **out**

Какие объекты вероятней всего не нужны?

- которые были созданы недавно
- которые созданы давно

Сборщик мусора

- Три поколения объектов:
  - 0 – Объект еще не переживал ни одной сборки мусора
  - 1 – Объект пережил одну сборку мусора
  - 2 – Объект пережил более одной сборки мусора
- Сборщик мусора запускается, если:
  - достигается пороговое значение объема памяти, занимаемого поколением 0
  - не хватает памяти
  - вызван явно
  - происходит выгрузка домена приложения
- Для определения объектов необходимых для удаления строит граф объектов. Проверяется доступность объектов от корней приложения. Объекты, на которые нет ссылок, - кандидаты на удаление
- Сборщик мусора сначала анализирует объекты поколения 0, затем 1, затем 2. Если после очистки объектов i-ого поколения памяти достаточно (не превышен порог i+1 поколения), то сборщик мусора остановится, если нет - займется следующим поколением.
- После очистки i-ого поколения, у выживших объектов (i-ого поколения) поколение увеличивается на 1

- Таким образом, сначала очищаются короткоживущие объекты. Объекты уровня приложения проверяются на возможность удаления редко

*Большие объемы данных сразу помечаются поколением 2 и помещаются в Ledge Object Heap (LOH). Более 85000 байт

Финализируемые объекты

- В классе **object** есть метод
  - protected virtual void Finalize()
- **Finalize()** нельзя переопределить. Компилятор пишет его сам, если в классе присутствует деструктор
  - protected override void Finalize() {

```
try{ // Специальный код сборщика мусора
    // Вызов деструктора
    // Специальный код сборщика мусора}
finally {
    base.Finalize();
    // Специальный код сборщика мусора}}
```

- Деструктор:
  - Синтаксис: *~Имя_класса() {…}*
  - Не допускает модификаторов
  - Всегда не более одного
  - Из деструктора нельзя обращаться к управляемым ресурсам самого объекта (и вообще к управляемым ресурсам), поскольку их может уже не существовать
  - Только для удаления неуправляемых ресурсов
  - Не может быть определен в структуре
- **Finalize()** вызывает сборщик мусора перед удалением объекта
- Точнее….

Сборщик мусора

- Если в классе реализован деструктор, то при сборке мусора, если объект помечен для удаления, сборщик мусора добавляет ссылку на объект из кучи в специальную очередь объектов, доступных для финализации.
- Объекты в этой таблице тоже являются рутами. Объект останется в куче не удаленным
- Специальный поток(-и) с высоким приоритетом разбирает очередь финализируемых объектов и вызывает метод **Finalise()** у каждого объекта
- После финализации объекта ссылка на него удаляется из очереди финализируемых объектов
- Только при следующей сборке мусора и в случае завершения финализации объекта, объект будет удален из кучи

- Для каждого финализируемого объекта требуется минимум 2 процесса сборки мусора
- Серьезные накладные расходы, для финализируемых объектов
- Деструктор используется только для удаления неуправляемых ресурсов
- ***Для удаления неуправляемых ресурсов есть более удачный подход***

Никогда не используйте деструктор *

* - Бывают случаи, когда деструктор необходим. Один из них – рекомендуемый шаблон реализации **IDisposable**, рассмотренный далее

Высвобождаемые объекты

- Реализуют интерфейс **IDisposable**

```
public interface IDisposable  {
        void Dispose();
}
```

- В реализации метода **Dispose**() необходимо очистить неуправляемые ресурсы
- Сборщик мусора ничего не знает о методе **Dispose**(), поэтому все объекты в очищаемом объекте еще существуют и к ним можно обращаться
- В методе **Dispose**() можно очищать управляемые ресурсы.
  - Если объект содержит другие объекты, которые реализуют интерфейс **IDisposable**, то лучше вызвать у них метод **Dispose**(), для того, чтобы они очистили свои ресурсы.
  - Вызовите **Dispose**() у базового класса (если он реализует **IDisposable**)
  - Установить **null** для управляемых ресурсов
- Метод **Dispose**() обычно реализуется так, чтобы его можно было вызывать несколько раз

- Если вы используете объект, класс которого реализует интерфейс **IDisposable**, то по окончанию работы с ним вызовите у него метод **Dispose**()
- Если вы используете неуправляемые ресурсы, то реализуйте интерфейс **IDisposable** и в методе **Dispose**() очистите все (главным образом неуправляемые) ресурсы.

Демонстрация

Реализация IDisposable

using

- Применяется только к **IDisposabe** объектам
- Синтаксис:

```
using (инициализация объекта)
{
        // использование объекта
}
```

- При выходе из тела блока **using** автоматически будет вызван метод **Dispose()**
- Пример:

```
using (StreamReader reader = new StreamReader(@"Program.cs"))
{
    Console.WriteLine(reader.ReadToEnd());
}
```

- При возникновении исключительной ситуации метод **Dispose()** все равно будет вызван
- Аналогичен блоку

```
StreamReader reader;
try{
    reader = new StreamReader(@"Program.cs") ;
   …}
filnally{    reader.Dispose();    }
```

Рекомендуемый шаблон IDisposable

```
public class MyResource : IDisposable
  {
      private bool disposed = false;
      public void Dispose()
      {
          Dispose(true);
          GC.SuppressFinalize(this);
      }
      protected virtual void Dispose(bool disposing)
      {
          if (!this.disposed)
          {
              if (disposing)
              {
                      // Удаление управляемых ресурсов. Вызов Dispose() у используемых объектах в полях
              }
              // Очистка неуправляемых ресурсов

              disposed = true;
          }
      }
      ~MyResource()
      {
          Dispose(false);
      }
  }
```

Garbage Collection (GC)

- System.GC
- Позволяет взаимодействовать со сборщиком мусора
- Статические методы:
  - Collect() – Заставляет сборщик мусора провести сборку мусора
    - Можно указать поколение для очистки
    - Можно указать режим сборки
    - Collect(int generation, GCCollectionMode mode)
  - CollectionCount(generation) – показывает сколько сборок мусора пережило старшее поколение
  - SuppressFinalize(object) – позволяет установить флаг, показывающий, что для данного объекта не должен вызываться его метод Finalize().
  - ReRegisterForFinalize(object) – обратный к SuppressFinalize()
  - GC.WaitForPendingFinalizers() – останавливает текущий поток, до тех пор пока финализирующий поток не финализирует все накопившиеся для финализации объекты
  - GC.KeepAlive(object) – оставит живым объект до этого вызова

Демонстрация

Реализация рекомендуемого шаблона IDisposable