

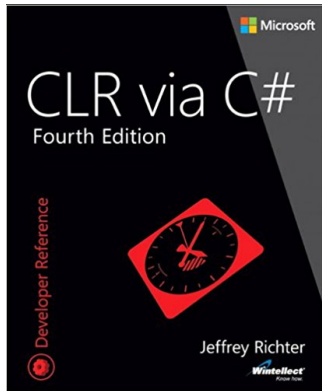
# Сборка мусора в .NET

Юрий Литвинов  
y.litvinov@spbu.ru

19.05.2023

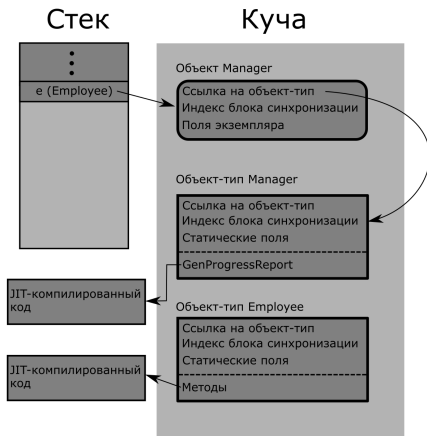
# Книжка

Все примеры и картинки взяты из J. Richter, CLR via C# (4th edition), Microsoft Press, 2012, 896pp



# Как объекты .NET представляются в памяти

```
let Example () =
    let e: Employee =
        new Manager()
    e.GenProgressReport()
```



# Работа с ресурсами

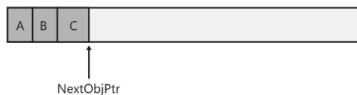
- ▶ Выделить память под объект, представляющий ресурс
- ▶ Инициализировать ресурс
- ▶ Использовать ресурс, вызывая методы объекта
- ▶ Деинициализировать ресурс
- ▶ Освободить память

В отличие от C++, .NET не даст испортить память (без **unsafe**) и уменьшает вероятность утечек памяти

# Выделение ресурсов

## Managed Heap

- ▶ Вычисление размера объекта
  - ▶ Поля объекта + поля, унаследованные от предка
- ▶ Добавление служебных данных
  - ▶ Указатель на объект-тип
  - ▶ Sync block index
- ▶ Проверка наличия свободного места на куче
- ▶ Сброс памяти в 0, вызов конструктора, возврат указателя на объект из **new**
  - ▶ Прямо перед возвратом продвигается указатель начала свободного места в куче

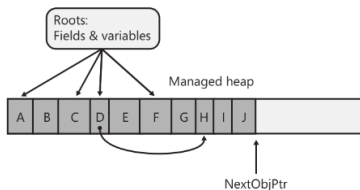


# Особенности

- ▶ Managed Heap выделяется при запуске процесса, но может расти
  - ▶ Доступно порядка 1.5Гб для 32-битных систем и порядка 8Тб для 64-битных
- ▶ Выделение памяти — просто прибавить размер объекта к указателю
  - ▶ Делается мгновенно, в отличие от C++
- ▶ Объекты лежат друг за другом подряд
  - ▶ Занимают возможно мало страниц памяти
  - ▶ С большей вероятностью помещаются в кеш процессора

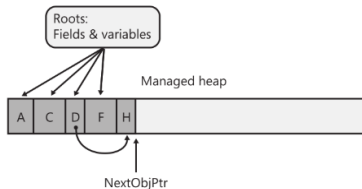
# Сборка мусора, mark

- ▶ **roots** — переменные ссылочных типов, лежащие на стеке
- ▶ **marking** — пометка всех объектов, достижимых из roots, как ЖИВЫХ
  - ▶ Исполнение всех потоков прерывается
  - ▶ Все объекты помечаются как мёртвые
    - ▶ Бит в sync block index
  - ▶ Гонится обход графа от root-ов



## Сборка мусора, sweep

- ▶ “Сжатие кучи” — все живые объекты сдвигаются в куче так, чтобы занимать непрерывный участок памяти
  - ▶ Восстанавливается locality of reference
  - ▶ Исключается фрагментирование памяти
    - ▶ Чтобы выделение можно было и дальше осуществлять просто сдвигом указателя
- ▶ Ссылки в живых объектах редактируются с учётом сдвигов
- ▶ Потоки продолжают выполнение
- ▶ Если после сборки мусора памяти всё ещё не хватает, из new бросается OutOfMemoryException
- ▶ Статические поля живут вечно





# Пример

Работает только на C# и только в .NET Framework

```
public static class Program {  
    public static void Main() {  
        // Create a Timer object that knows to call our TimerCallback  
        // method once every 2000 milliseconds.  
        Timer t = new Timer(TimerCallback, null, 0, 2000);  
        // Wait for the user to hit <Enter>  
        Console.ReadLine();  
    }  
  
    private static void TimerCallback(Object o) {  
        // Display the date/time when this method got called.  
        Console.WriteLine("In TimerCallback: " + DateTime.Now);  
        // Force a garbage collection to occur for this demo.  
        GC.Collect();  
    }  
}
```

## Так тоже будет плохо

```
public static class Program {  
    public static void Main() {  
        // Create a Timer object that knows to call our TimerCallback  
        // method once every 2000 milliseconds.  
        Timer t = new Timer(TimerCallback, null, 0, 2000);  
        // Wait for the user to hit <Enter>  
        Console.ReadLine();  
        // Refer to t after ReadLine (this gets optimized away)  
        t = null;  
    }  
  
    private static void TimerCallback(Object o) {  
        // Display the date/time when this method got called.  
        Console.WriteLine("In TimerCallback: " + DateTime.Now);  
        // Force a garbage collection to occur for this demo.  
        GC.Collect();  
    }  
}
```

## А вот так будет ок

```
public static class Program {  
    public static void Main() {  
        // Create a Timer object that knows to call our TimerCallback  
        // method once every 2000 milliseconds.  
        Timer t = new Timer(TimerCallback, null, 0, 2000);  
        // Wait for the user to hit <Enter>  
        Console.ReadLine();  
        // Refer to t after ReadLine (t will survive GCs until Dispose returns)  
        t.Dispose();  
    }  
  
    private static void TimerCallback(Object o) {  
        // Display the date/time when this method got called.  
        Console.WriteLine("In TimerCallback: " + DateTime.Now);  
        // Force a garbage collection to occur for this demo.  
        GC.Collect();  
    }  
}
```

# Не всё так просто

## Поколения

Каждый раз чистить память во всей куче было бы дорого, поэтому придумали поколения:

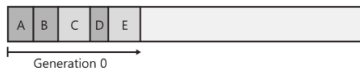
- ▶ Чем новее объект, тем меньше, скорее всего, он будет нужен
  - ▶ Различного рода локальные переменные и переменные внутри тел циклов
- ▶ Чем дольше объект живёт, тем дольше, скорее всего, он ещё проживёт
- ▶ Собрать память в части кучи можно быстрее, чем во всей куче

Все новые объекты добавляются в поколение 0:

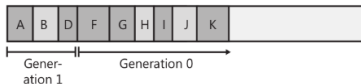


# Первая сборка мусора

- ▶ Под поколение 0 выделяется “бюджет” памяти
- ▶ При переполнении поколения 0 происходит сборка мусора и все выжившие объекты перемещаются в поколение 1
  - ▶ Поколение 0 после GC всегда пусто

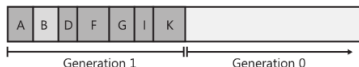


- ▶ Следующие выделения идут в поколение 0
- ▶ Объекты в поколении 1 могут стать недостижимы



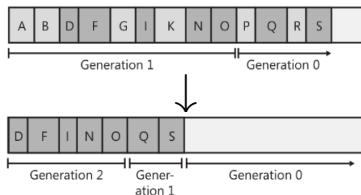
# Последующие сборки мусора

- ▶ Сборки мусора в Gen0 игнорируют Gen1, пока бюджет первого поколения не превышен
  - ▶ При JIT-е для записи в поле ссылочного типа генерируется Write Barrier
  - ▶ Write Barrier выставляет флаг в Card Table, если объект в поколении 1 или 2
  - ▶ Сборщик мусора смотрит на Card Table, не обходя объекты из Gen1 и Gen2
  - ▶ Если есть изменения, то честно гонится обход и объекты помечаются как живые
  - ▶ После сборки мусора Card Table сбрасывается
  - ▶ Несколько медленнее запись в поля ссылочных типов, но значительно быстрее сборка мусора
- ▶ Недостижимые объекты в Gen1 могут пережить сборку мусора
- ▶ Сборка мусора в поколении 0 сравнима по времени с page fault (меньше миллисекунды)



## Что дальше

Если бюджет поколения 1 превышен, запускается сборка мусора и выжившие объекты перемещаются в поколение 2:



Больше поколений не бывает (GC.MaxGeneration возвращает 2)  
Сборщик мусора динамически управляет бюджетами на поколения

- ▶ Если выживших в поколении мало, бюджет уменьшается
- ▶ Сборки мусора происходят чаще, но быстрее, и надо меньше памяти
- ▶ В реактивных системах часто всё Gen0 — мусор

## Когда происходит сборка мусора

- ▶ Закончился бюджет на поколение 0
- ▶ Явный вызов `System.GC.Collect`
  - ▶ Настоятельно не рекомендуется
- ▶ ОС сообщает о нехватке памяти
  - ▶ Да, CLR подписывается на события ОС
- ▶ Выгружается `AppDomain`
  - ▶ Выполняется финализация всех объектов
- ▶ Нормально завершается процесс с приложением
  - ▶ В отличие от убийства процесса через Task Manager
  - ▶ Выполняется только финализация, сжатие кучи не делается



# Large Object Heap

- ▶ Объекты размером больше примерно 85000 байт живут по особым правилам
- ▶ Отдельная куча для таких объектов
- ▶ Не происходит сжатие кучи
  - ▶ Возможна фрагментация
  - ▶ Возможен `OutOfMemoryException` из-за фрагментации
- ▶ Они всегда считаются в поколении 2
  - ▶ Короткоживущие большие объекты заставляют GC страдать

## Режимы сборки мусора

- ▶ Workstation — частая быстрая сборка на одном ядре, чтобы не тормозить подолгу и не мешать другим приложениям
  - ▶ По умолчанию
- ▶ Server — многопоточная сборка, занимающая весь процессор (считается, что приложение на сервере одно)
  - ▶ На одноядерной машине бесполезен и не используется

App.config/Web.config, включающий серверный GC:

```
<configuration>  
  <runtime>  
    <gcServer enabled="true"/>  
  </runtime>  
</configuration>
```

## Многопоточная сборка

- ▶ Не Server-режим!
- ▶ Mark выполняется в отдельном потоке во время работы приложения
- ▶ Если требуется сборка поколений 0 или 1, всё работает как обычно
- ▶ Если требуется сборка поколения 2, в поколении 0 выделяется объект вне бюджета и приложение продолжает работать
- ▶ В это время гонится mark по поколению 2 в отдельном потоке
- ▶ Когда закончили, останавливаем потоки и выполняем sweep
- ▶ Mark уже пометил большинство достижимых объектов, так что сборка будет гораздо быстрее
- ▶ Сжатие кучи, как правило, при этом не происходит, если памяти ещё много

App.config/Web.config, **выключающий** параллельный GC:

```
<configuration>  
  <runtime>  
    <gcConcurrent enabled="false"/>  
  </runtime>  
</configuration>
```

# Динамическая настройка GC

- ▶ Workstation/Server режимы выставляются при запуске и их нельзя менять
- ▶ Можно менять `GCSettings.GCLatencyMode`:
  - ▶ **Batch** — выключает параллельный GC
  - ▶ **Interactive** — включает параллельный GC (включён по умолчанию для Workstation)
  - ▶ **LowLatency** — избегание сборки поколения 2 (только по явному запросу или исчерпанию системной памяти)
  - ▶ **SustainedLowLatency** — избегание сборки поколения 2, минимизация сборок
    - ▶ Настолько близко к реальному времени, насколько .NET может

# Особенности

- ▶ В любой непонятной ситуации кидается `OutOfMemoryException`
  - ▶ `LowLatency` имеет смысл включать по возможности ненадолго
  - ▶ `Constrained execution region` для переключения
  - ▶ Настройки GC действуют на весь процесс
    - ▶ Менять настройки GC из нескольких потоков сразу — плохая идея

```
private static void LowLatencyDemo() {  
    GCLatencyMode oldMode = GCSettings.LatencyMode;  
    System.Runtime.CompilerServices.RuntimeHelpers.PrepareConstrainedRegions();  
    try {  
        GCSettings.LatencyMode = GCLatencyMode.LowLatency;  
        // Run your code here...  
    }  
    finally {  
        GCSettings.LatencyMode = oldMode;  
    }  
}
```

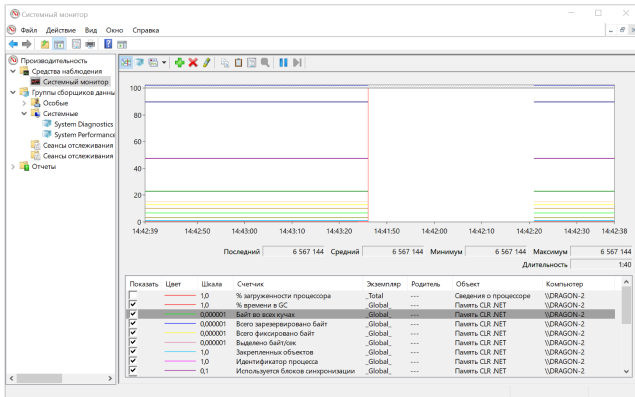
# Ручная сборка

- ▶ GC.Collect
  - ▶ Поколение, до которого собирать
  - ▶ Режим
  - ▶ Параллельная или блокирующая сборка
- ▶ Режим:
  - ▶ **Default** — то же, что Forced (может поменяться)
  - ▶ **Forced** — принудительная сборка всех поколений до и включая запрошенное
  - ▶ **Optimized** — сборка, если GC считает это разумным
- ▶ В общем случае лучше не применять, бывает полезно, если в приложении есть *не повторяющееся* событие, освобождающее кучу объектов и нам как можно скорее надо почистить память (зачем бы то ни было)
- ▶ Collect обновляет бюджеты поколений

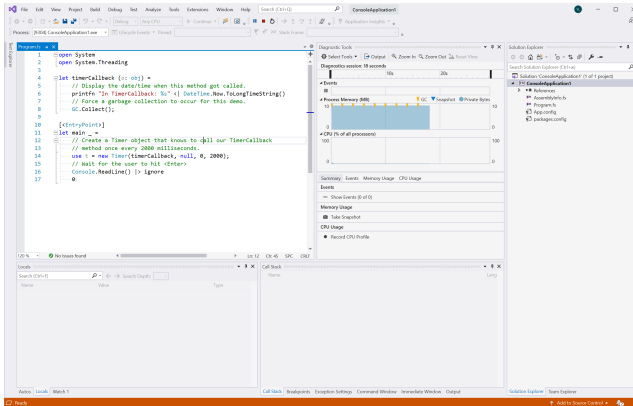
# Мониторинг

Int32 `CollectionCount`(Int32 generation);  
 Int64 `GetTotalMemory`(Boolean forceFullCollection);

- Performance Counters (PerfMon.exe -> Add Counters -> .NET CLR Memory)



# Мониторинг





# Финализаторы

- ▶ Если объекту требуется только память, то сборщик мусора всё сделает за нас, но бывает так, что требуются ещё системные ресурсы
  - ▶ `System.IO.FileStream`, `System.Threading.Mutex`
- ▶ Метод *Finalize()*
- ▶ Финализатор (просто синтаксис для *Finalize*):

```
internal sealed class SomeType {  
    ~SomeType() {  
        ...  
    }  
}
```

Не деструктор! *Finalize* вызывается, “когда хочет”

## Дьявол в деталях

- ▶ Объекты с финализаторами всегда переживают сборку в своём поколении
  - ▶ Сборщик мусора должен дать финализатору отработать
  - ▶ Все поля объекта с финализатором тоже живут дольше, чем надо
- ▶ Порядок вызова финализаторов не определён
  - ▶ Из финализатора нельзя обращаться к объекту с финализатором
- ▶ Финализаторы запускаются в отдельном потоке
  - ▶ Нельзя рассчитывать на определённый поток
  - ▶ Если `Finalize` повис, то всё, все финализируемые объекты будут жить вечно
  - ▶ Если `Finalize` бросил исключение, то весь процесс аварийно завершается

# System.Runtime.InteropServices.SafeHandle

```

public abstract class SafeHandle : CriticalFinalizerObject, IDisposable {
    protected IntPtr handle;

    protected SafeHandle(IntPtr invalidHandleValue, Boolean ownsHandle);
    protected void SetHandle(IntPtr handle);
    public void Dispose() { Dispose(true); }
    protected virtual void Dispose(Boolean disposing) {
        // The default implementation ignores the disposing argument.
        // If resource already released, return
        // If ownsHandle is false, return
        // Set flag indicating that this resource has been released
        // Call virtual ReleaseHandle method
        // Call GC.SuppressFinalize(this) to prevent Finalize from being called
        // If ReleaseHandle returned true, return
        // If we get here, fire ReleaseHandleFailed Managed Debugging Assistant (MDA)
    }
    ~SafeHandle() { Dispose(false); }
    protected abstract Boolean ReleaseHandle();
...
    public void DangerousAddRef(ref Boolean success) {...}
    public IntPtr DangerousGetHandle() {...}
    public void DangerousRelease() {...}
}

```

# Особенности

- ▶ **CriticalFinalizerObject**
  - ▶ Все методы `Finalize` в иерархии наследования компилируются в момент создания объекта
  - ▶ Их `Finalize` вызываются в последнюю очередь
  - ▶ Вызываются даже при аварийной выгрузке `AppDomain`-а
- ▶ `handle` — что-то вроде указателя на ресурс, невалиден, если равен 0 или -1
  - ▶ `SafeHandleZeroOrMinusOneIsInvalid`
- ▶ Про `SafeHandle` знает компилятор, поэтому:
  - ▶ может обрабатывать в него результат нативного вызова
  - ▶ умеет считать количество передач `SafeHandle` в нативный код

# Пример, SafeFileHandle

```
public sealed class SafeFileHandle : SafeHandleZeroOrMinusOnesInvalid {  
    public SafeFileHandle(IntPtr preexistingHandle, Boolean ownsHandle)  
        : base(ownsHandle) {  
        base.SetHandle(preexistingHandle);  
    }  
  
    protected override Boolean ReleaseHandle() {  
        // Tell Windows that we want the native resource closed.  
        return Win32Native.CloseHandle(base.handle);  
    }  
}
```

# IDisposable, так не работает

```
public static class Program {  
    public static void Main() {  
        // Create the bytes to write to the temporary file.  
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };  
  
        // Create the temporary file.  
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);  
  
        // Write the bytes to the temporary file.  
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);  
  
        // Delete the temporary file.  
        File.Delete("Temp.dat"); // Throws an IOException  
    }  
}
```

# IDisposable, так работает

```
public static class Program {  
    public static void Main() {  
        // Create the bytes to write to the temporary file.  
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };  
  
        // Create the temporary file.  
        FileStream fs = new FileStream("Temp.dat", FileMode.Create);  
  
        // Write the bytes to the temporary file.  
        fs.Write(bytesToWrite, 0, bytesToWrite.Length);  
  
        // Explicitly close the file when finished writing to it.  
        fs.Dispose();  
  
        // Delete the temporary file.  
        File.Delete("Temp.dat"); // Throws an IOException  
    }  
}
```

## IDisposable, детали

- ▶ Если хоть одно поле IDisposable, то и весь класс должен быть IDisposable
- ▶ Класс, реализующий IDisposable, должен уметь бросать `System.ObjectDisposedException` из всех своих методов и свойств
- ▶ `Dispose` не должен бросать исключение
  - ▶ Если вызван несколько раз, то просто ничего не делать
- ▶ Если вы явно вызываете `Dispose` из своего кода, вы, скорее всего, делаете что-то не так
- ▶ `Dispose` не обязан быть thread-safe



# using

```
public static class Program {  
    public static void Main() {  
        // Create the bytes to write to the temporary file.  
        Byte[] bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };  
        // Create the temporary file.  
        using (FileStream fs = new FileStream("Temp.dat", FileMode.Create)) {  
            // Write the bytes to the temporary file.  
            fs.Write(bytesToWrite, 0, bytesToWrite.Length);  
        }  
        // Delete the temporary file.  
        File.Delete("Temp.dat");  
    }  
}
```

► Или **using var** в C# 8 или **use** в F#

## Очередной способ прострелить себе ногу

```
FileStream fs = new FileStream("DataFile.dat", FileMode.Create);  
StreamWriter sw = new StreamWriter(fs);  
sw.Write("Hi there");
```

*// The following call to Dispose is what you should do.*

```
sw.Dispose();
```

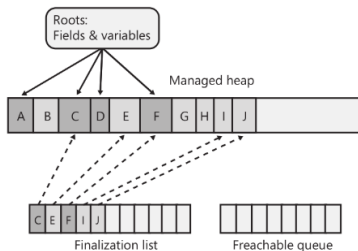
*// NOTE: StreamWriter.Dispose closes the FileStream;*

*// the FileStream doesn't have to be explicitly closed.*

Как реализовать финализаторы FileStream и StreamWriter, если порядок их вызова не определён?

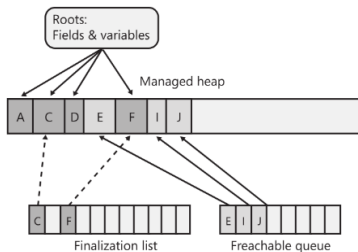
# Финализация изнутри

- ▶ Finalization list — туда добавляются все объекты, переопределяющие `Finalize()`, в момент создания
- ▶ Freachable queue — туда перекладываются объекты из Finalization list, которые должны умереть, в момент сборки мусора



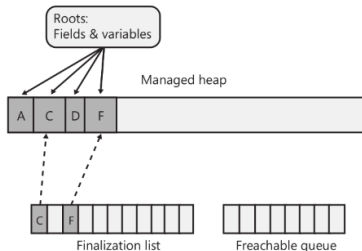
## Финализация изнутри (2)

- ▶ Объекты из Freachable queue считаются сборщиком мусора root-ами и не умирают
- ▶ Финализаторы запускаются в отдельном потоке для каждого объекта из очереди
- ▶ Объекты выкидываются из очереди в процессе



## Финализация изнутри (3)

- ▶ При сборке мусора в следующем поколении финализированные объекты собираются, потому что на них уже точно нет ссылок



# Ручное управление жизнью объекта

- ▶ `System.Runtime.InteropServices.GCHandle`
  - ▶ Value type, содержит указатель на запись в handle table
  - ▶ Записи в handle table содержат объекты, но не все они считаются root-ами
- ▶ `GCHandleType`:
  - ▶ **Weak** — позволяет узнать, что объект уже не нужен, но не факт, что финализатор отработал
  - ▶ **WeakTrackResurrection** — позволяет узнать, что объект уже не нужен и уже финализирован
  - ▶ **Normal** — просьба к GC держать объект в памяти, но можно двигать
  - ▶ **Pinned** — просьба к GC держать объект в памяти и не двигать
    - ▶ Ключевое слово **fixed**

# Пример, fixed

```

unsafe public static void Go() {
    // Allocate a bunch of objects that immediately become garbage
    for (Int32 x = 0; x < 10000; x++) new Object();
    IntPtr originalMemoryAddress;
    Byte[] bytes = new Byte[1000]; // Allocate this array after the garbage objects

    // Get the address in memory of the Byte[]
    fixed (Byte* pbytes = bytes) { originalMemoryAddress = (IntPtr) pbytes; }

    // Force a collection; the garbage objects will go away & the Byte[] might be compacted
    GC.Collect();

    // Get the address in memory of the Byte[] now & compare it to the first address
    fixed (Byte* pbytes = bytes) {
        Console.WriteLine("The Byte[] did{0} move during the GC",
            (originalMemoryAddress == (IntPtr) pbytes) ? " not" : null);
    }
}

```

# WeakReference

```
public sealed class WeakReference<T> : ISerializable where T : class {  
    public WeakReference(T target);  
    public WeakReference(T target, Boolean trackResurrection);  
    public void SetTarget(T target);  
    public Boolean TryGetTarget(out T target);  
}
```