|  |
| --- |
| , RD Dep. |
| Конспект и раздаточный материал  NET.C#.08 Управление ресурсами в .NET. Сборка мусора |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial version | Анжелика Кравчук |  |  |  |
| 1.1 | Review and corrections. | Владимир Тихон |  |  |  |

Contents

[1. Урок 1. Введение в сборку мусора 3](#_Toc301377508)

[1.1. Жизненный цикл объекта 3](#_Toc301377509)

[1.2. Управляемые ресурсы в .NET Framework 4](#_Toc301377510)

[1.3. Как работает сборщик мусора? 6](#_Toc301377511)

[1.4. Класс GC 10](#_Toc301377512)

[1.5. Демонстрация: Класс GC 12](#_Toc301377513)

[1.6. Определение деструктора 12](#_Toc301377514)

[1.7. Демонстрация: Реализация деструктора 15](#_Toc301377515)

[2. Урок 2. Управление ресурсами 16](#_Toc301377516)

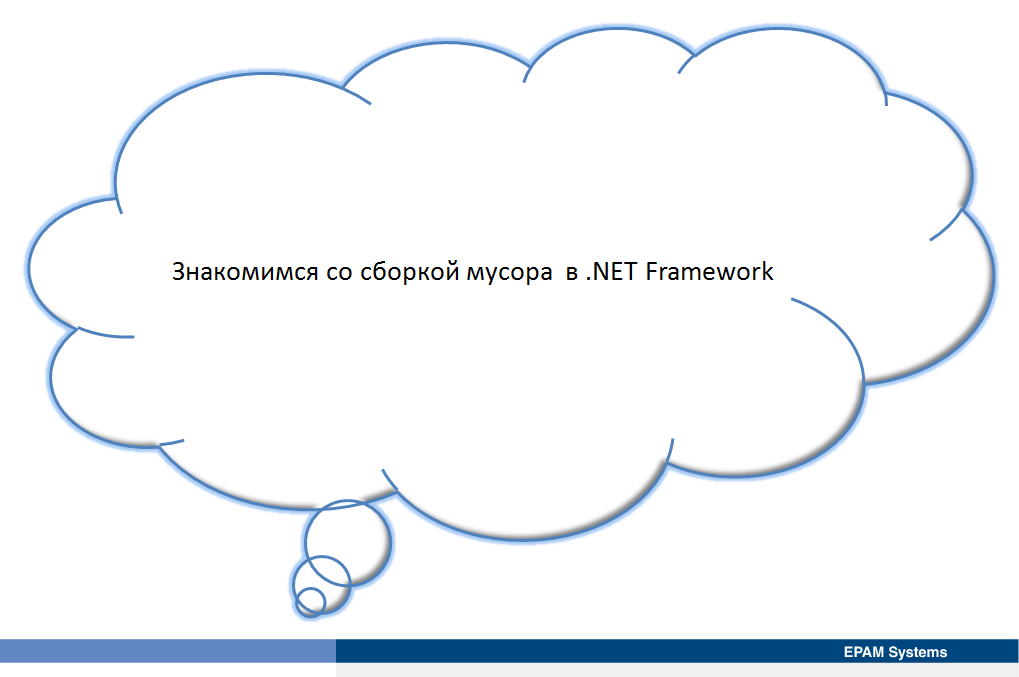
[2.1. Зачем существует управление ресурсами в управляемой среде? 16](#_Toc301377517)

[2.2. Что такое шаблон dispose? 17](#_Toc301377518)

[2.3. Управление ресурсами в приложениях 21](#_Toc301377519)

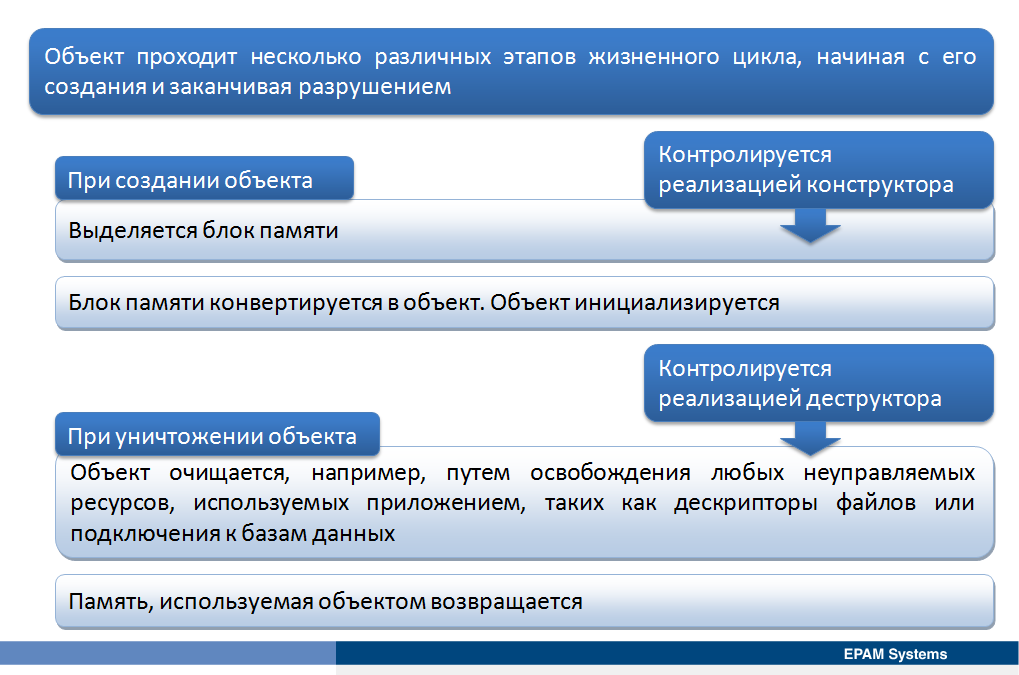
[2.4. Демонстрация: Использование шаблона dispose 23](#_Toc301377520)

# Урок 1. Введение в сборку мусора



Каждый объект, создаваемый в приложении, проходит жизненный цикл от создания до уничтожения. При уничтожении объекта, его состояние должно быть очищено, и любые используемые им управляемые ресурсы должны быть освобождены. В .NET Framework эти задачи выполняет сборщик мусора.

## Жизненный цикл объекта



Объект проходит несколько различных этапов жизненного цикла, начиная с его создания и заканчивая разрушением. Процесс, используемый программистом для создания объекта на первый взгляд чрезвычайно прост; для создания нового экземпляра объекта используется ключевое слово new. Однако, процесс, происходящий при создании объекта на самом деле не так прост, поскольку при этом происходит следующее:

* Выделяется блок памяти. Этот блок памяти достаточно большой, чтобы сохранять объект.
* Блок памяти конвертируется в объект. Объект инициализируется.

При этом можно контролировать только второй из этих двух шагов, превращающий блок памяти в объект. Этот шаг контролируется реализацией конструктора. CLR выполняет распределение памяти для управляемых объектов, однако, если вызываются неуправляемые библиотеки, возможно, потребуется вручную выделять память для их создания. После создания объекта можно использовать его свойства, методы и другие члены.

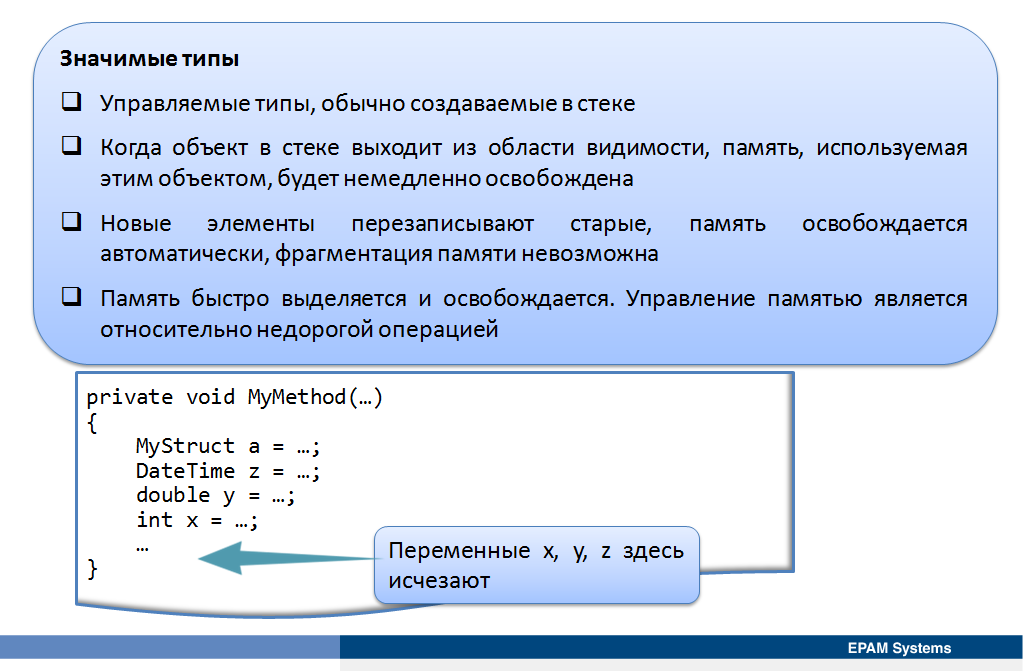
Когда работа с объектом завершена, он может быть уничтожен. Для возвращения любых ресурсов, используемых этим объектом, используется уничтожение (деструкция). Подобно созданию уничтожение это двухэтапный процесс:

* Объект очищается, например, путем освобождения любых неуправляемых ресурсов, используемых приложением, таких как дескрипторы файлов или подключения к базам данных.
* Память, используемая объектом возвращается.

Контролировать можно только первый из этих этапов – очистку объекта и освобождения ресурсов. Данный шаг можно регулировать за счет реализации деструктора.

CLR управляет освобождением памяти, используемой управляемыми объектами, однако, если используются неуправляемые объекты, может потребоваться вручную высвобождать память, используемую этими элементами.

## Управляемые ресурсы в .NET Framework



.NET Framework делит элементы, которые может использовать управляемое приложение, на две большие категории: значимые и ссылочные типы.

Значимые типы это управляемые типы, обычно создаваемые в стеке. CLR управляет стеком. Когда объект в стеке выходит из области видимости, память, используемая этим объектом, немедленно освобождается. Например, в конце метода любые определенные в нем переменные, основанные на значимом типе (созданные в стеке), уничтожаются. CLR поддерживает указатель на вершину стека. При созлании переменной значимого типа, она помещается на вершину стека, а указатель стека перемещается вверх. Когда переменная выходит из области видимости, указатель стека перемещается снова вниз. Таким образом, новые элементы перезаписывают старые и память освобождается автоматически; поэтому управление памятью является относительно недорогой операцией.

Ссылочные типы размещаются в куче. Куча это блок памяти, контролируемый CLR отдельно от стека. При создании объекта CLR выделяет память для объекта и создает ссылки на него в стеке. В отличие от значимого типа, ссылочный тип может иметь несколько ссылок на один и тот же объект. При этом если одна ссылка на объект исчезнет при выходе из области видимости, другие ссылки на этот объект могут все еще находится в области видимости и оставаться действующими. Объект может быть уничтожен, его деструктор сработает, а его ресурсы высвободяться только тогда, когда последняя ссылка на объект исчезнет. Следовательно, время жизни объекта не регулируется рамками какой-либо одной ссылки на этот объект.

Важно понять, что управляемая куча представляет собой нечто большее, чем просто случайный фрагмент памяти, к которому CLR получает доступ. Сборщик мусора .NET «убирает» кучу довольно тщательно, причем, при необходимости даже сжимает пустые блоки памяти с целью оптимизации. Чтобы ему было легче это делать, в управляемой куче поддерживается указатель (обычно называемый указателем на следующий объект или указателем на новый объект, NextObjPtr), который показывает, где точно будет размещаться следующий объект (Рис. 1).

**A**

**B**

**С**

Указатель на следующий объект

Рис. 1.

Важной функцией сборщика мусора .NET Framework является наблюдение за объектом в куче и определение, когда последняя ссылка на этот объект исчезнет, тогда объект может быть безопасно уничтожен. Определение момента, когда объект не имеет ссылки, может быть трудоемкой и дорогостоящей операцией, поэтому сборщик мусора выполняет эту задачу только тогда, когда это необходимо, как правило, когда количество доступной памяти в куче падает ниже некоторого порога.

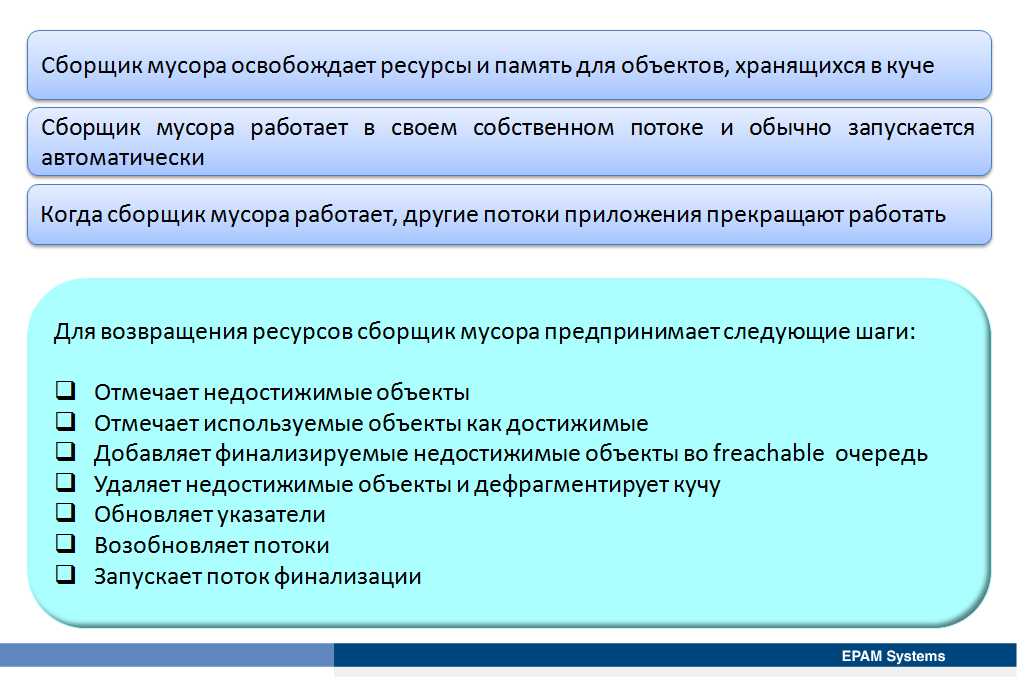
Следует отметить, что установка ссылки в null никоим образом не вынуждает сборщик мусора немедленно приступить к делу и удалить объект из кучи, а просто позволяет явно разорвать связь между ссылкой и объектом, на который она ранее указывала. Благодаря этому, присваивание ссылкам значения null в С# имеет гораздо меньше последствий, чем в других языках на базе С, и совершенно точно не причинит никакого вреда.

Второй функцией сборщика мусора является дефрагментация кучи. Если приложение пытается создать объект, для которого в настоящее время недостаточно смежного пустого места в куче, сборщик мусора будет пытаться переместить некоторые существующие объекты и сжать результируюшее свободное пространство в достаточно большой кусок памяти, чтобы сохранить новый объект. Опять же, следует отметить, что вычислительно это может быть очень дорогостоящей операцией.

Значимые типы обычно создаются в стеке. Однако, существует сценарий, когда это не так. Например, когда при разработке класса значимый тип используется в качестве его поля. При создании экземпляра объекта с помощью этого класса, поля, которые образовывают объект, в том числе любые поля значимого типа размещаются в куче. Эти поля значимого типа остаются активными до тех пор, пока содержащий их объект не разрушится, и сборщик мусора высвободит пространство, используемое ими.

Если значимый тип используется как поле в структуре, он будет сохранен как часть структуры. Структуры являются значимыми типами и в этом качестве, как правило, хранятся в стеке, если не являются элементами внутри класса. Если значимый тип является элементом класса, он хранится в куче, как описано ранее.

## Как работает сборщик мусора?



Сборщик мусора освобождает ресурсы и память для объектов, хранящихся в куче. Он работает в своем собственном потоке и обычно запускается автоматически при определенных обстоятельствах. Когда сборщик мусора работает, другие потоки приложения прекращают работать, поскольку сборщик мусора может перемещать объекты в памяти и должен обновить указатели на правильные адреса для перемещенных объектов.

Для возвращения ресурсов сборщик мусора предпринимает следующие шаги:

* Отмечает недостижимые объекты; объекты считаются недостижимым если не доказано иное.
* Начинает с объектов, на которые есть ссылки в стеке, и отмечает используемые объекты как достижимые. Он выполняет это рекурсивно, если объект, который уже отмечен как достижимый ссылается на другой объект, этот объект также отмечен как достижимый . Сборщик мусора включает логику, позволяющую предотвратить бесконечную рекурсию, например там, где есть циклические ссылки между двумя объектами.
* Проверяет, имеются ли какие-либо объекты, которые были помечены как недостижимые, деструкторы, которых должны быть выполнены. Выполнение деструктора называется финализацией. Любые объекты, которые требуют финализации перемещаются в структуру данных, поддерживаемую сборщиком мусора и называемую очередью объектов, доступных для финализации (freachable queue). Freachable очередь хранит указатели на объекты, требующие завершения до восстановления их ресурсов.
* Объекты, добавленные в freachable очередь, отмечены как достижимые, потому что существуют действительные ссылки на них; деструктор должен быть запущен перед тем, как их память может быть освобождена. Объекты, как правило, добавляется в freachable очередь только один раз.
* Объекты, отмеченные как достижимые перемещаются вниз кучи для формирования непрерывного блока, дефрагментируя кучу. Ссылки на объекты (в стеке и в других объектах в куче), перемещенные сборщиком мусора, обновляются.
* Другие потоки возобновляются.
* В отдельном потоке, объекты, добавленные в freachable очередь завершаются. После завершения объекта, указатель на этот объект удаляется из freachable очереди. Объекты, не удаляются из памяти до следующего раза работы сборщика мусора.

Каким образом сборщик мусора определяет момент, когда объект уже более не нужен? Чтобы разобраться в стоящих за этим деталях, необходимо знать, что собой представляет корневые элементы приложения (application roots). Корневым элементом (root) называется ячейка в памяти, в которой содержится ссылка на размещающийся в куче объект. Строго говоря, корневыми могут называться элементы любой из перечисленных ниже категорий:

* Ссылки на любые статические объекты или статические поля.
* Ссылки на локальные объекты в пределах кодовой базы приложения.
* Ссылки на передаваемые методу параметры объектов.
* Ссылки на объекты, ожидающие финализации.

Во время процесса сборки мусора исполняющая среда будет исследовать объекты в управляемой куче, чтобы определить, являются ли они по-прежнему достижимыми (т.е. корневыми) для приложения. Для этого среда CLR будет создавать графы объектов, представляющие все достижимые для приложения объекты в куче. При этом сборщик мусора никогда не будет создавать граф для одного и того же объекта дважды, избегая необходимости выполнения подсчета циклических ссылок.

Чтобы увидеть все это на примере, предположим, что в управляемой куче содержится набор объектов с именами A, B, C, D, E, F и G. Во время сборки мусора эти объекты (а также любые внутренние объектные ссылки, которые они могут содержать) будут исследованы на предмет наличия у них активных корневых элементов. После построения графа все недостижимые объекты (в примере - объекты С и F) помечаются как являющиеся мусором. Граф объектов в только что описанном сценарии (линии со стрелками следует воспринимать как «зависит от» или «требует»; например, «Е зависит от G и В», «А не зависит ни от чего» и т.д.) выглядит следующим образом (Рис. 2).

Управляемая куча

A

B

C

D

E

F

G

NextObjPtr

A

B

D

E

G

Рис. 2.

После того как объекты помечены для уничтожения (объекты С и F в графе объектов во внимание не принимаются), они будут удалены из памяти (Рис. 3). Оставшееся пространство в куче будет после этого сжиматься до компактного состояния, что, в свою очередь, вынудит CLR изменить набор активных корневых элементов приложения (и лежащих в их основе указателей) так, чтобы они ссылались на правильное место в памяти. И, наконец, указатель на следующий объект тоже будет подстраиваться так, чтобы указывать на следующий доступный участок памяти.

Управляемая куча

A

B

D

E

G

NextObjPtr

Рис. 3.

При размещении и удалении объектов CLR использует ряд оптимизаций.

Во-первых, сборщик мусора использует две отдельных кучи, одна из которых предназначена специально для хранения очень больших объектов (объекты размером более 85 Кб ). Доступ к этой куче во время сборки мусора осуществляется реже из-за возможных последствий в плане производительности, в которые может выливаться изменение места размещения больших объектов. Невзирая на этот факт, управляемая куча все равно может спокойно считаться единой областью памяти.

Во-вторых, управляемая куча для малых объектов выделяет три поколения объектов.

При попытке обнаружить недостижимые объекты CLR не проверяет буквально каждый находящийся в куче объект. Очевидно, что на это уходила бы масса времени. Для оптимизации процесса каждый объект в куче относится к определенному поколению. Смысл в применении поколений выглядит довольно просто: чем дольше объект находится в куче, тем выше вероятность того, что он там и будет оставаться. С другой стороны, объекты, которые были размещены в куче лишь недавно (как, например, те, что находятся в пределах области действия метода), вероятнее всего будут становиться недостижимым довольно быстро. Исходя из этих предположений, каждый объект в куче относится к одному из перечисленных ниже поколений.

* Поколение 0. Идентифицирует новый только что размещенный объект, который еще никогда не помечался как подлежащий удалению в процессе сборки мусора.
* Поколение 1. Идентифицирует объект, который уже "пережил" один процесс сборки мусора (был помечен как подлежащий удалению в процессе сборки мусора, но не был удален из-за наличия достаточного места в куче).
* Поколение 2. Идентифицирует объект, которому удалось пережить более одного прогона сборщика мусора.

**A**

**B**

**C**

**D**

**E**

Поколение 0

Вновь инициализируемая куча, содержащая некоторые объекты, все поколения 0. Еще не было сборки мусора.

**A**

**B**

**D**

Поколение 1

Поколение 0

После первой сборки мусора: выжившие объекты из поколения 0 попали в поколение 1; поколение 0 пусто.

**A**

**B**

**D**

Поколение 1

Поколение 0

**F**

**G**

**H**

**I**

**J**

**K**

Созданы новые объекты в поколении 0; в поколени 1 появились «мусорные» объекты.

**A**

**B**

**D**

Поколение 1

Поколение 0

**F**

**G**

**I**

**K**

После двух сборок мусора: выжившие объекты из поколения 0 попали в поколение 1 (растущий размер поколения 1); поколение 0 пусто.

**A**

**B**

**D**

Поколение 1

Поколение 0

**F**

**G**

**I**

**K**

**L**

**M**

**N**

**O**

Новые объекты появляются в поколении 0; поколение 1 имеет больше «мусорных» объектов.

**A**

**B**

**D**

Поколение 1

Поколение 0

**F**

**G**

**I**

**K**

**N**

**O**

Через три сборки мусора: выжившие объекты из поколения 0 попали в поколение 1 (растущий размер поколения  1); поколение 0 пусто.

**A**

**B**

**D**

Поколение 1

Поколение 0

**F**

**G**

**I**

**K**

**N**

**O**

**P**

**Q**

**R**

**S**

Новые объекты выделяются в поколении 0; поколение 1 имеет больше «мусорных» объектов.

**D**

Поколение 2

Поколение 0

**F**

**I**

**N**

**O**

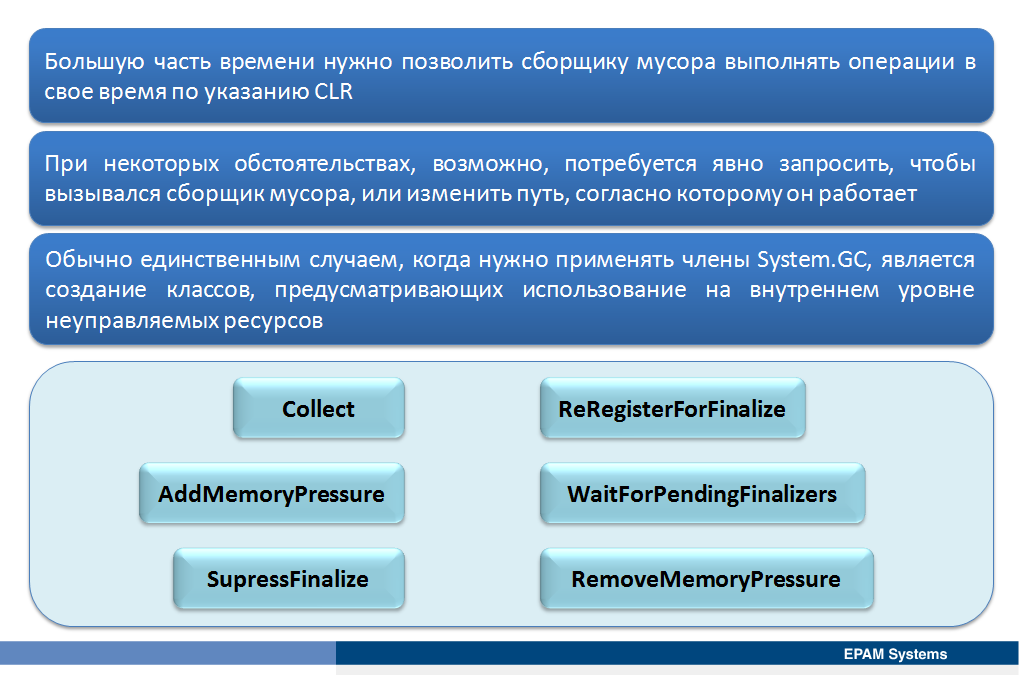
**Q**

**S**

Поколение 1

Через четыре сборки мусора: выжившие в поколение 1 попали в поколение 2, выжившие из поколения 0 попали в поколение 1, поколение 0 пусто.

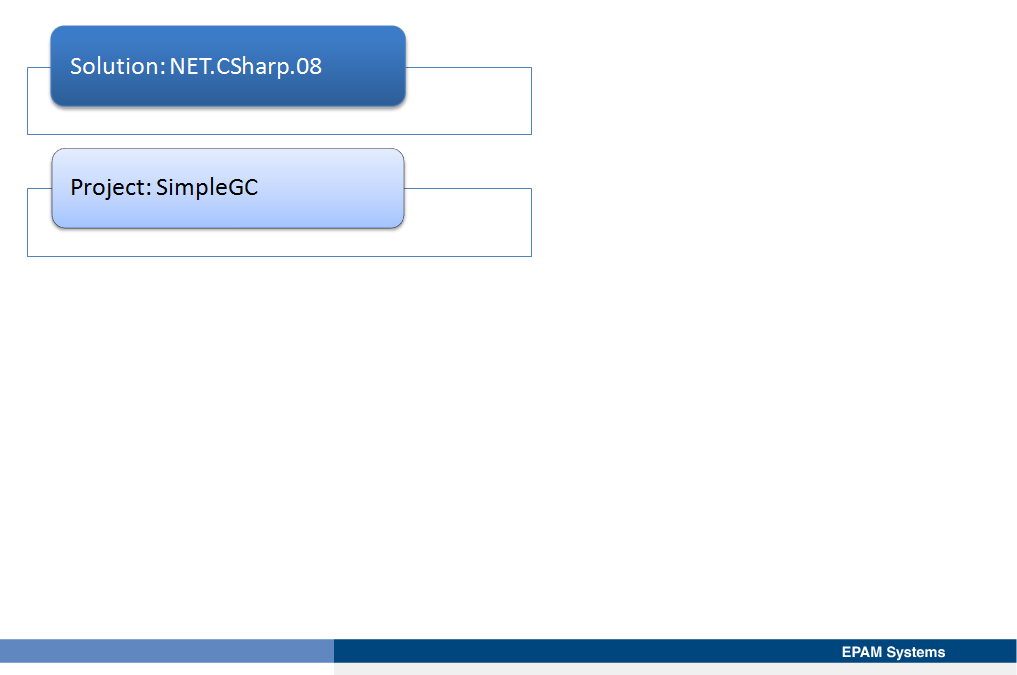
## Класс GC



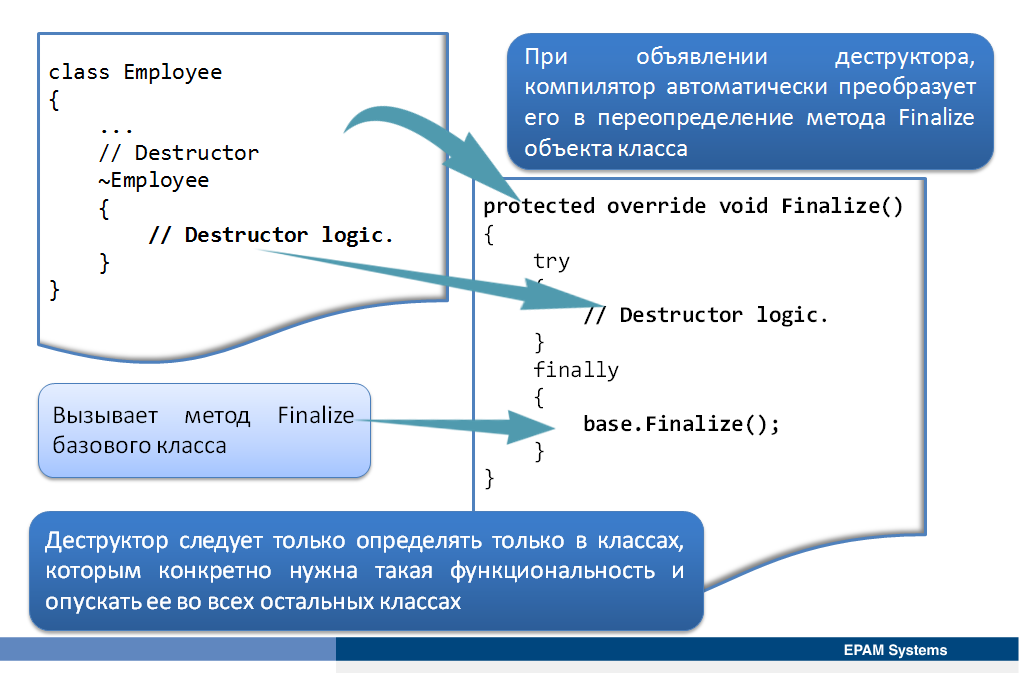
Большую часть времени нужно позволить сборщику мусора выполнять операции в свое время по указанию CLR. Однако, при некоторых обстоятельствах, возможно, потребуется явно запросить, чтобы вызывался сборщик мусора, или изменить путь, согласно которому он работает. Обычно единственным случаем, когда нужно применять члены System.GC, является создание классов, предусматривающих использование на внутреннем уровне неуправляемых ресурсов. Для этого, можно использовать класс GC. Класс GC включает в себя несколько статических методов, которые можно вызывать из кода. Следующая таблица включает в себя некоторые из наиболее часто используемых методов, предоставляемых классом GC.

|  |  |  |
| --- | --- | --- |
| **Метод** | **Описание** | **Примечания** |
| **Collect** | Форсирует сборку мусора. | Следует избегать использования метода Collect в коде. Если заставлять сборщик мусора запускаться чаще, чем это необходимо, это может иметь отрицательное влияние на производительность приложения.  Метод Collect является асинхронным, когда он возвращается, нет никакой гарантии, что сборка мусора завершена, или даже начата, известно только то, что сборщик мусора будет работать в следующий подходящий интервал.  GC.Collect(); |
| **WaitForPendingFinalizers** | Приостанавливает текущий поток до тех пор, пока все объекты в freachable очереди не будут завершены. | Этот метод использется, если нужно специально подождать завершения для всех объектов, находящихся в настоящее время в freachable очереди.  GC.WaitForPendingFinalizers(); |
| **SupressFinalize** | Предотвращает завершение объекта, переданного в качестве параметра. | Метод вызывается при реализации шаблона dispose. Использование метода может повысить производительность, предотвращая от выполения дважды кода завершения.  GC.SuppressFinalize(this); |
| **ReRegisterForFinalize** | Запрашивает финализатор для объекта, который либо уже завершен или завершение было подавлено. | Метод используется, если есть подавленый для завершения объект, или объект уже завершен, но требуется выполнения, чтобы завершить объект снова.  GC.ReRegisterForFinalize(this); |
| **AddMemoryPressure** | Информирует исполняющую среду о выделении большого объема неуправляемой памяти, которую необходимо учесть при планировании сборки мусора. | Этот метод сообщает исполняющей среде, что будет выделятся большой блок памяти, и он будет освобождать ресурсы, где это возможно. При использовании этого метода, следут указать, сколько памяти необходимо выделить. Если нужно выделить несколько блоков памяти, можно вызывать метод в приложении несколько раз. Необходимо вызвать этот метод до выделения большого блока неуправляемой памяти. Не слудует использовать этот метод при создании управляемых объектов.  GC.AddMemoryPressure(1000); |
| **RemoveMemoryPressure** | Информирует исполняющую среду, что высвобожден большой блок неуправляемой памяти и ее более не требуется учитывать при планировании сборки мусора. | Этот метод сообщает исполняющей среде, что вы высвободили большой блок памяти, и это позволит снизить срочность, с которой она выполняет сбор мусора. При использовании этого метода, вы должны указать, какой объем памяти вы высвободили. Если необходимо удалить несколько блоков памяти, вы можете вызывать метод в вашем приложении несколько раз. Вы должны вызвать этот метод, после высвобождения большого блока неуправляемой памяти. Вы не должны использовать этот метод, если вы разрушаете управляемые объекты. Вам следует всегда использовать методы AddMemoryPressure и RemoveMemoryPressure вместе, чтобы гарантировать, что можно добавлять и удалять точно такое же количество памяти.  GC.RemoveMemoryPressure(1000); |

## Демонстрация: Класс GC



## Определение деструктора



Дальнейшее исследование процесса сборки мусора продолжается на примере создания финализируемых (finalizable) и высвобождаемых (disposable) объектов. Очень важно отметить, что описанные далее приемы подходят в случае построения управляемых классов, внутри которых используются неуправляемые ресурсы.

В базовом классе .NET System.Object имеется виртуальный метод Finalize(). В предлагаемой по умолчанию реализации он ничего особенного не делает. За счет его переопределения в специальных классах устанавливается специфическое место для выполнения любой необходимой данному типу логики по очистке. Из-за того, что метод Finalize() по определению является защищенным (protected), вызывать его напрямую из класса экземпляра с помощью операции точки не допускается. Вместо этого метод Finalize () (если он поддерживается) будет автоматически вызываться сборщиком мусора перед удалением соответствующего объекта из памяти.

Вызов метода Finalize() будет выполняться (в конечном итоге) либо во время естественной активизации процесса сборки мусора, либо во время его принудительной активизации программным образом с помощью GC. Collect().

В подавляющем большинстве классов С# необходимость в создании явной логики по очистке или специального финализатора возникать не будет. Объясняется это очень просто: если в классах используются лишь другие управляемые объекты, все они рано или поздно все равно будут подвергаться сборке мусора. Единственным случаем, когда может возникать потребность в создании класса, способного выполнять после себя процедуру очистки, является работа с неуправляемыми ресурсами (такими как низкоуровневые файловые дескрипторы, низкоуровневые неуправляемые соединения с базами данных, фрагменты неуправляемой памяти и т.п.).

Момент, в который работает деструктор, явно не указан, и нельзя гарантировать порядок, в котором будут работать деструкторы для различных объектов. Поэтому, не следует делать каких-либо предположений и вводить зависимостей между объектами в деструкторе.

Чтобы определить деструктор следует добавить знак тильды (~), а затем указать имя класса. Затем определить логику деструктора в фигурных скобках. В следующем примере показан синтаксис для добавления деструктора.

class Employee

{

...

// Destructor

~Employee

{

// Destructor logic.

}

}

К деструкторам применяются следующие ограничения:

* Нельзя добавить деструктор структуре или любому другому типу значения. Типы значения хранятся в стеке, так что сборщик мусора к ним не применяется.
* Нельзя объявить модификатор доступа для деструктора (являются неявно защищенными). Деструкторы всегда вызываются только сборщиком мусора, и их нельзя вызывать напрямую из кода.
* Нельзя объявить деструктор, который принимает параметры. Сборщик мусора вызывает деструктор, и нет возможности его контролировать, передавая ему параметры.
* Класс может иметь только один деструктор.
* Деструкторы не могут наследоваться или перегружаться.

При объявлении деструктора, компилятор автоматически преобразует его в переопределение метода Finalize объекта класса, однако нельзя переопределить метод Finalize самостоятельно. Нужно объявить деструктор, и компилятор выполняет преобразование. Деструктор в предыдущем примере компилятор преобразует для переопределения метода Finalizeследующим образом.

protected override void Finalize()

{

try

{

// Destructor logic.

}

finally

{

base.Finalize();

}

}

Компилятор добавляет логику из деструктора блоку try, а затем вызывает метод Finalize базового класса в блоке finally. Это гарантирует выполнение метода Finalize базового класса всегда. даже если код гененрирует исключение.

Деструктор следует только определять только в классах, которым конкретно нужна такая функциональность и опускать ее во всех других классах.

При размещении объекта в управляемой куче исполняющая среда автоматически определяет, поддерживается ли в нем какой-нибудь специальный метод Finalize(). Если да, тогда она помечает его как финализируемый (finalizable) и сохраняет указатель на него во внутренней очереди, называемой список финализации (finalizatlon list) (Рис. 4). Этот список финализации представляет собой просматриваемую сборщиком мусора таблицу, где перечислены объекты, которые перед удалением из кучи должны быть обязательно финализированы.

**A**

**B**

**C**

**D**

**E**

**F**

**G**

**H**

**I**

**J**

Управляемая куча (managed heap)

**C**

**E**

**F**

**I**

**J**

Список финализации (finalization list)

Очередь объектов для финализации (freachable queue)

Корневые элементы

(strong references)

Глобальные

Статические

Локальные

Регистры процессора

Рис. 4.

Когда сборщик мусора определяет, что наступило время удалить объект из памяти, он проверяет каждую запись в очереди финализации и копирует объект из кучи в еще одну управляемую структуру, называемую таблицей объектов, доступных для финализации (freachable queue) (Рис. 5). После этого он создает отдельный поток для вызова метода Finalize в отношении каждого из упоминаемых в этой таблице объектов при следующей сборке мусора. В результате получается, что для окончательной финализации объекта требуется как минимум два процесса сборки мусора (Рис. 6).

**A**

**C**

**D**

**E**

**F**

**I**

**J**

Управляемая куча (managed heap)

**C**

**F**

Список финализации (finalization list)

Очередь объектов для финализации (freachable queue)

Корневые элементы

(strong references)

Глобальные

Статические

Локальные

Регистры процессора

**E**

**I**

**J**

Рис. 5.

**A**

**C**

**D**

**F**

Управляемая куча (managed heap)

**C**

**F**

Список финализации (finalization list)

Очередь объектов для финализации (freachable queue)

Корневые элементы

(strong references)

Глобальные

Статические

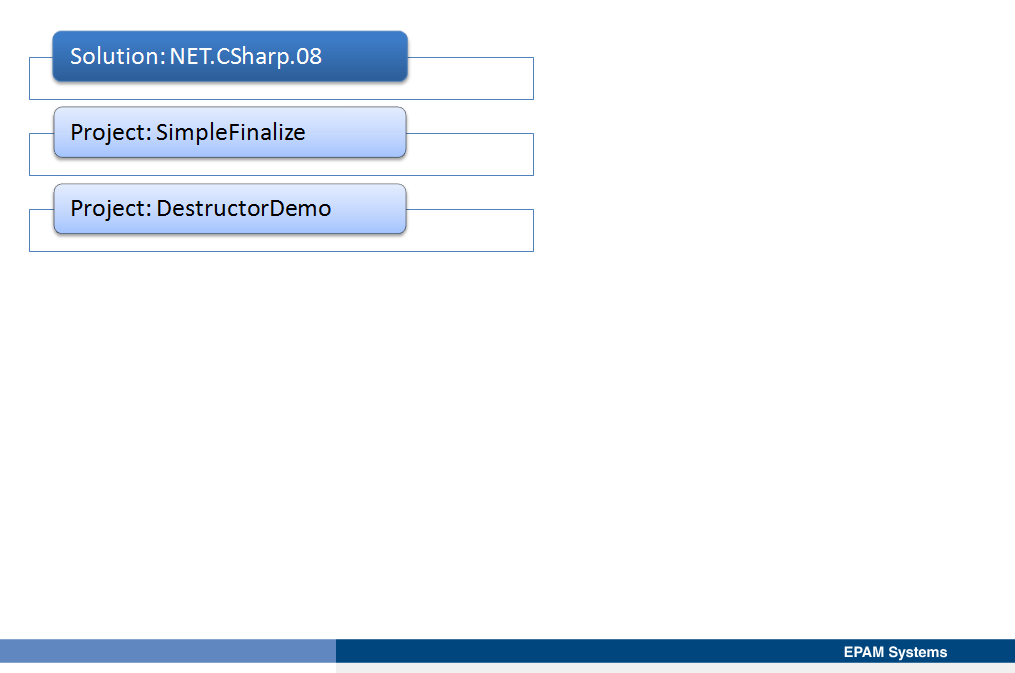
Локальные

Регистры процессора

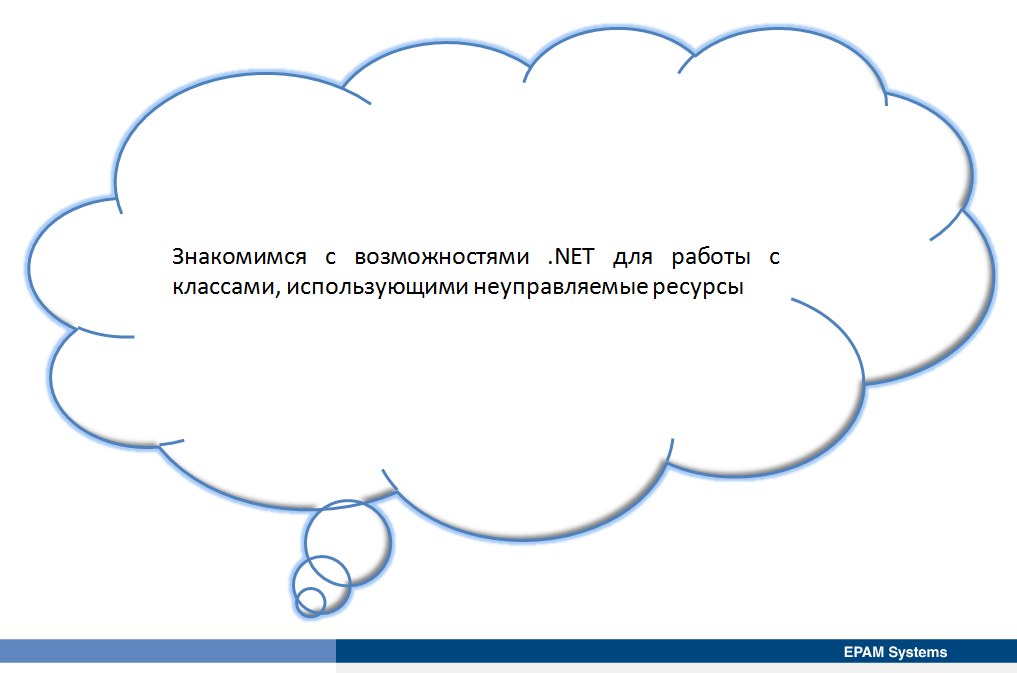
Рис. 6.

<http://go.microsoft.com/fwlink/?LinkId=192944>

## Демонстрация: Реализация деструктора

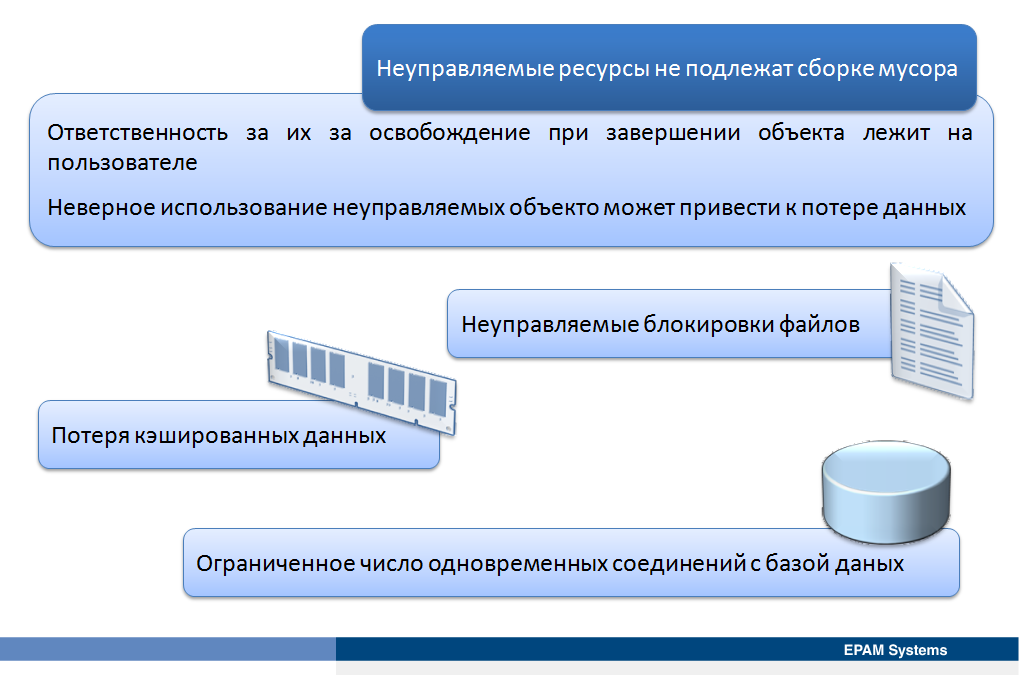


# Урок 2. Управление ресурсами



Сборщик мусора автоматически восстанавливает память и ресурсы для управляемых объектов. Однако, если в классе используются неуправляемые ресурсы, следует принять дополнительные меры для обеспечения того, чтобы они высвободились надлежащим образом. Паттерн dispose является шаблоном проектирования, позволяющим высвободить неуправляемые ресурсы, используемые классом, контролируемо и своевременно. Реализация паттерна dispose в типе способствуют тому, что приложения будут работать хорошо и не будут сохранять неуправляемые ресурсы, дольше, чем это необходимо.

## Зачем существует управление ресурсами в управляемой среде?



Сборщик мусора связан с управляемыми объектами. Он не знает, как освободить ресурсы, связанные с неуправляемыми объектами. Если в классе существует ссылка на неуправляемые ресурсы, при удалении последней ссылки на класс, неуправляемый объект не будет уничтожен. Операционная система не сможет очистить ресурсы до тех пор, пока приложение не завершится.

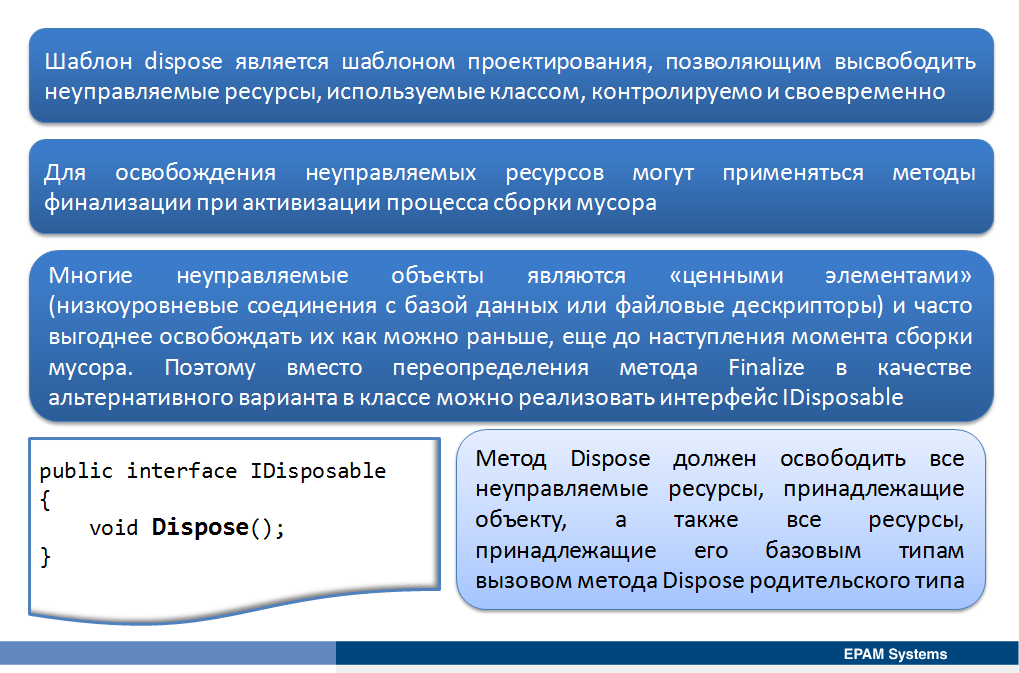
Библиотека классов .NET Framework, обеспечивает, например, класс TextWriter, используемый для открытия файла локальной файловой системы и записи текста в файл. Класс TextWriter действует как управляемая оболочка вокруг текстовых файлов, являющихся неуправляемыми ресурсами, контролируемыми операционной системой. Когда объект TextWriter открывает файл, операционная система блокирует файл, чтобы никакие другие процессы не смогли писать в этот файл. Когда использование объекта в коде завершено TextWriter, можно удалить все ссылки на него. Это действие уничтожит управляемый объект TextWriter, но не сможет освободить блокировку, потому что это часть неуправляемого ресурса, не контролируемая сборщиком мусора. Необходимо предпринять дополнительные меры с целью освобождения этой блокировки, в противном случае, если создать другой объект TextWriter для записи в этот же файл будет невозможно.

В дополнение к неуправляемым блокировкам, существует несколько других проблем, связанных с неправильным управлением ресурсами. Например, некоторые неуправляемые типы используют буферы памяти для повышения производительности и записывают в базовый источник данные только тогда, когда буфер либо полон, либо флеширован. Если при уничтожении объекта не флешировать такой буфер, его содержимое может быть потеряно. Так, например, при записи данных в файл с помощью класса TextWriter, данные могут быть буферизированы базовым файловым типом. Если объект TextWriter уничтожить без высвобождения ресурсов, связанных с файлом, буфер может быть флеширован не правильно, и возможна потеря данных. Класс TextWriter предоставляет для записи содержимого буфера в файл метод Flush, который можно вызвать, чтобы все данные записались в файл до уничтожения объекта TextWriter.

Соединения с базой данных это другой ресурс, который и дорог в обслуживании, и часто ограничен. Серверы баз данных часто поддерживают только ограниченное число одновременных соединений. Если не высвободитеь соединения с базой данных при завершении ее использования, доступные соединения с базой данных скоро исчерпаются, и приложение может выбросить неожиданные исключения при попытке подключения к базе данных.

Если правильно управлять ресурсами, обеспечивая, чтобы все неуправляемые ресурсы освобождались, когда они больше не требуется, можно предотвратить все описанные выше проблемы.

## Что такое шаблон dispose?



Шаблон dispose является шаблоном проектирования, позволяющим высвободить неуправляемые ресурсы, используемые классом, контролируемо и своевременно. Реализация в типе этого шаблона будет способствовать тому, что приложения будут хорошо работать, и не сохранять неуправляемые ресурсы дольше, чем это необходимо. .NET Framework. предоставляя интерфейс IDisposable и объект, реализующий этот интерфейс, должен следовать этому паттерну.

Для освобождения неуправляемых ресурсов могут применяться методы финализации при активизации процесса сборки мусора. Однако многие неуправляемые объекты являются «ценными элементами» (низкоуровневые соединения с базой данных или файловые дескрипторы) и часто выгоднее освобождать их как можно раньше, еще до наступления момента сборки мусора. Поэтому вместо переопределения метода Finalize в качестве альтернативного варианта в классе можно реализовать интерфейс IDisposable.

Интерфейс IDisposable определяет единственный метод Dispose, не принимающий никаких параметров. Метод Dispose должен освободить все неуправляемые ресурсы, принадлежащие объекту. Он также должен освободить все ресурсы, принадлежащие его базовым типам вызовом метода Dispose родительского типа.

public interface IDisposable

{

void Dispose();

}

Когда реализуется поддержка интерфейса IDisposable, предполагается, что после завершения работы с объектом метод Dispose должен вручную вызываться пользователем этого объекта, прежде чем объектной ссылке будет позволено покинуть область видимости. Благодаря этому объект может выполнять любую необходимую очистку неуправляемых ресурсов без попадания в очередь финализации и без ожидания того, когда сборщик мусора запустит содержащуюся в классе логику финализации.

Метод Dispose отвечает не только за освобождение неуправляемых ресурсов типа, но и за вызов аналогичного метода в отношении любых других содержащихся в нем высвобождаемых объектов. В отличие от метода Finalize, в нем вполне допустимо взаимодействовать с другими управляемыми объектами. Объясняется это очень просто: сборщик мусора не имеет понятия об интерфейсе IDisposable и потому никогда не будет вызывать метод Dispose. Следовательно, при вызове данного метода пользователем объект будет все еще существовать в управляемой куче и иметь доступ ко всем остальным находящимся там объектам.

class MyResourceWrapper: IDisposable

{

public void Dispose()

{

//Освобождение неуправляемых ресурсов. . .

//Избавление от других содержащихся внутри

//и пригодных для очистки объектов.

//Только для целей тестирования

Console.WriteLine("\*\*\*\*\*In Dispose!\*\*\*\*\*");

}

}

. . .

MyResourceWrapper rw = new MyResourceWrapper();

rw.Dispose();

Многие из классов .NET Framework, оборачивающих неуправляемые ресурсы, такие как класс TextWriter, реализуют интерфейс IDisposable. При создании собственных классов, ссылаемых на неуправляемые типы, необходимо реализовать интерфейс IDisposable. Для любого создаваемого напрямую объекта, если он поддерживает интерфейс IDisposable, следует всегда вызывать метод Dispose.

Некоторые из типов библиотек базовых классов, реализуя интерфейс IDisposable, предусматривают использование псевдонима для метода Dispose, чтобы заставить отвечающий за очистку метод звучать более естественно для типа, в котором он определяется. Например, класс System.I0.FileStream реализует интерфейс IDisposable и, следовательно, поддерживает метод Dispose, но при этом в классе определяется метод Close, применяемый для той же цели.

FileStream fs = new FileStream("myFile.txt", FileMode.OpenOrCreate);

Fs.Close();

Fs.Dispose();

Вызов метода Dispose не разрушает объект, он остается существовать и после выполнения метода Dispose; объект уничтожается только после того, как окончательная ссылка на него удаляется, а сборщик мусора восстанавливает все ресурсы, используемые им. Таким образом, при реализации в классе паттерна dispose необходимо отслеживать статус удаляемого объекта и проверять был ли метод Dispose уже вызван, а ресурсы высвобождены. Распространенным методом является добавление в класс поля isDisposed типа Boolean, установка его в методе Dispose и проверка в любом другом методе класса. Если методы в классе вызываются после удаления объекта, необходимо генерировать исключение ObjectDisposedException.

Исключением из этого правила является собственно метод Dispose. Необходимо иметь возможность запускать метод Dispose несколько раз без выбрасывания любых исключений или получения противоречивого состояния. Метод Dispose должен включать логику, необходимую для проверки состояния ресурсов, которые вот-вот будут освобождены, до их освобождения. В следующем примере показан класс, реализующий интерфейс IDisposable и включающий в себя объект TextWriter. Метод Dispose в этом примере используется для того, чтобы объект TextWriter был правильно закрыт, и основные ресурсы файла могли повторно использоваться.

class LogFileWriter : IDisposable

{

private bool isDisposed = false;

private TextWriter writer = ...;

...

public void WriteDataToFile(...)

{

// Check that the current object has not been disposed of

if (isDisposed)

{

throw new ObjectDisposedException(...);

}

...

}

public void Dispose()

{

if (!isDisposed)

{

// Only close the TextWriter if it is not null

// (in which case it has already been disposed)

if (writer != null)

{

writer.Flush();

writer.Close();

writer = null;

}

// Indicate that the object has been disposed of and

// resources have been released

isDisposed = true;

}

}

}

К этому моменту были рассмотрены два различных подхода, которые можно применять для создания класса, способного производить очистку и освобождать неуправляемые ресурсы. Первый подход заключается в переопределении метода System.Object.Finalize и позволяет гарантировать, что объект будет очищаться во время процесса сборки мусора (когда бы тот не запускался) без вмешательства со стороны пользователя. Второй подход предусматривает реализацию интерфейса IDisposable и позволяет обеспечить пользователя объекта возможностью очищать объект сразу же по окончании работы с ним. Однако если пользователь забудет вызвать метод Dispose, неуправляемые ресурсы могут остаться в памяти на неопределенный срок.

Если нужно гарантировать, чтобы метод Dispose вызывался всегда, можно включить его в качестве части процесса завершения, выполняемого сборщиком мусора. Для этого, можно добавить деструктор для своего класса и вызвать в нем метод Dispose. Однако следует помнить, что финализация является потенциально дорогостоящим процессом, поэтому реализовывать эту стратегию следует только тогда, когда это действительно необходимо.

В некоторых случаях можно распоряжаться управляемыми ресурсами в дополнение к неуправляемым ресурсам. Это необходимо, как правило, в случае, когда управляемый ресурс больше не требуется и дорог в обслуживании, например, большой массив. Сборщик мусора в конце концов вернет эту память, когда объект будет уничтожен, но можно попытаться освободить память, используемую этим массивом раньше, установив ссылку на этот массив в null в методе Dispose. Следует обратить внимание, что эта стратегия не гарантирует, что память, используемая массивом, будут возвращена раньше, чем могла бы. Время зависит от сборщика мусора.

Если из деструктора вручную вызывается метод Dispose, в дополнение к возможности приложения вызывать метод Dispose, нет смысла пытаться избавиться от управляемых ресурсов больше, чем один раз. В этом случае, рекомендуемый подход является перегрузкой метода Dispose и обеспечивает реализацию, которая принимает логический флаг, указывающий, был ли метод Dispose вызван в рамках завершения процесса или непосредственно в коде приложения. Обыкновением является передать значение true, если метод Dispose вызывается приложением и false, если он вызывается деструктором.

Перегруженный метод Dispose должен распоряжаться только управляемыми ресурсами, как будто бы он был вызван непосредственно (параметром является true). Если параметр false, управляемые ресурсы либо уже были удалены либо будут удалены сборщиком мусора в любом случае. В этом случае, метод Dispose должен лишь попытаться освободить неуправляемые ресурсы. Открытый метод Dispose, который не принимает параметров и определяется как часть интерфейса IDisposable, можно просто вызвать Dispose(true), а деструктор может вызвать Dispose(false).

Хорошей практикой является сделать перегруженную реализацию метода Dispose protected и virtual. Таким образом, метод может быть доступен коду в классе и всем его дочерним классам, но дочерний класс может переопределить его, если он определяет дополнительные ресурсы, которые должны быть освобождены. Перегруженный метод Dispose должен также вызвать метод Dispose любого класса родителя, если родительский класс реализует шаблон dispose.

В следующем примере кода показан класс, использующий описанную стратегию.

class LogFileWriter : ..., IDisposable

{

private bool isDisposed = false;

private TextWriter writer = ...;

private int largeArray[] = ...;

...

public void WriteDataToFile(...)

{

// Check that the current object has not been disposed of

if (isDisposed)

{

throw new ObjectDisposedException(...);

}

...

}

public void Dispose()

{

Dispose(true);

}

~LogFileWriter()

{

Dispose(false);

}

protected virtual void Dispose(bool isDisposing)

{

if (!isDisposed)

{

if (isDisposing)

{

// Release managed resources only if Dispose

// was called by the application

largeArray = null;

...

}

// Always release unmanaged resources

if (writer != null)

{

writer.Flush();

writer.Close();

writer = null;

}

// Indicate that the object has been disposed of and

// resources have been released

isDisposed = true;

// Call Dispose in the parent class

// (assuming that the parent provides this method)

base.Dispose(isDisposing);

}

}

}

При добавлении деструктора в класс, по умолчанию, когда сборщик мусора управляет ресурсами, объекты добавляются в freachable очередь для финализации. В public методе Dispose, после того как высвобождены все необходимые ресурсы, следует вызвать статический метод SuppressFinalize класса GC, передавая текущий объект, для того, чтобы сборщик мусора не тратил время выполняя код завершения для объекта, который уже очищен.

public void Dispose()

{

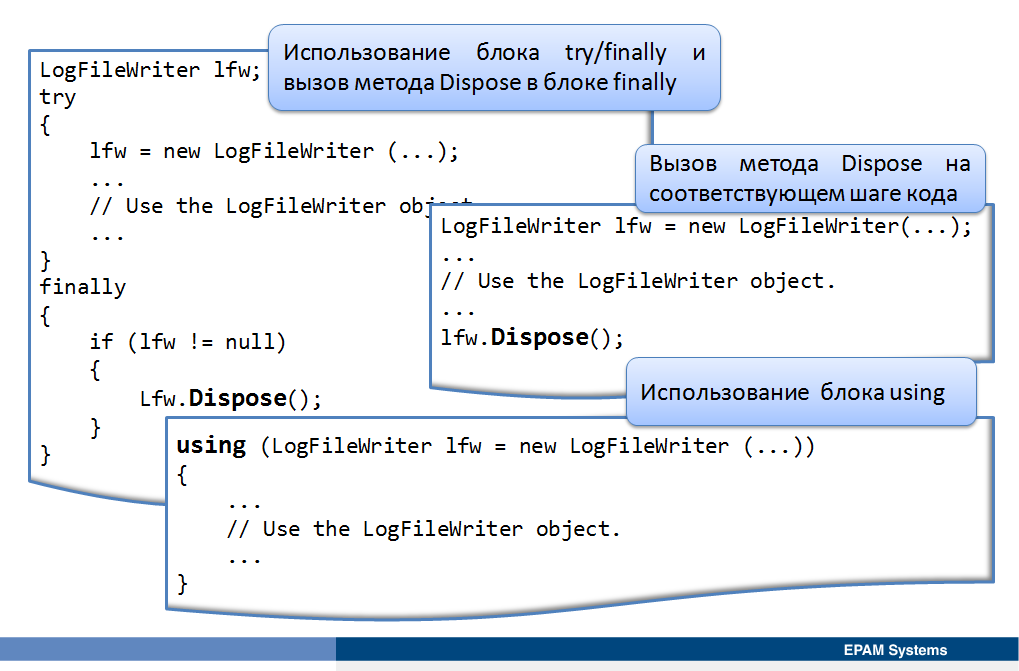
Dispose(true);

GC.SuppressFinalize(this);

}

<http://go.microsoft.com/fwlink/?LinkId=192946>

## Управление ресурсами в приложениях



Простое использование типов, реализующих интерфейс IDisposable не является достаточным для управления ресурсами; нужно помнить о вызове метода Dispose в коде. Существует несколько подходов, которые можно использовать, чтобы распоряжаться объектом, когда он больше не нужен:

* Можно вручную вызвать метод Dispose на соответствующем шаге кода.

LogFileWriter lfw = new LogFileWriter(...);

...

// Use the LogFileWriter object.

...

lfw.Dispose();

* Можно использовать блок try/finally и ликвидировать объект в блоке finally.

LogFileWriter lfw;

try

{

lfw = new LogFileWriter (...);

...

// Use the LogFileWriter object.

...

}

finally

{

if (lfw != null)

{

Lfw.Dispose();

}

}

* Можно использовать блок using (не следует путать с ключевым словом using для импорта пространтсва имен) для инкапсуляции удаляемых объектов.

using (LogFileWriter lfw = new LogFileWriter (...))

{

...

// Use the LogFileWriter object.

...

}

Использование блока using (третий вариант) является предпочтительным способом, чтобы гарантировать, что объект удаляется, когда закончено его использование. При добавлении в код блока using объявленные в нем переменные доступны только в этом блоке. Блок using является исключительно безопасным, что означает, что если код в блоке генерирует исключение, среда будет по-прежнему распоряжаться объектами, указанными в объявлении using.

Чтобы определить блок using, необходимо указать ключевое слово using и фигурные скобки, внутри которых объявляется и инициализируется переменная, используемая в блоке. Затем добавляетеся код, заключенный в фигурные скобки после объявления using. В следующем примере определяется блок using, который объявляет и инициализирует переменную LogFileWriter.

using (LogFileWriter lfw = new LogFileWriter (...))

{

...

// Use the LogFileWriter object.

...

}

Приведенный выше пример функционально эквивалентен следующему коду.

{

LogFileWriter lfw;

try

{

lfw = new LogFileWriter (...);

...

// Use the LogFileWriter object.

...

}

finally

{

if (lfw != null)

{

lfw.Dispose();

}

}

}

## Демонстрация: Использование шаблона dispose

