IT-Academy

ОБРАЗОВАТЕЛЬНЫЙ ЦЕНТР
ПРОГРАММИРОВАНИЯ И ВЫСОКИХ ТЕХНОЛОГИЙ

2020г.

Сборка мусора

- ✓ Механизм сборки мусора в .NET
- ✓ Поколения объектов
- ✓ Утечка памяти
- ✓ Финализатор и уничтожение объекта
- ✓ Работа со сборщиком мусора из кода. Класс GC
- ✓ Паттерн Dispose

Значимые и ссылочные типы

Значимые типы – создаются на стеке.
+ : более быстрая работа, автоматическое уничтожение ресурсов
- : малый размер стековой памяти

Ссылочные типы – создаются в управляемой куче.
+ : большой объем рабочей памяти
- : сложности при работе, связанные с освобождением используемых ресурсов

При использовании переменных типов значений, все значения этих переменных попадают в стек. После завершения работы стек очищается.

При использовании ссылочных типов, для них также будет отводиться место в стеке, но там будет храниться не значение, а адрес на участок памяти в куче, в котором и будут находиться сами значения данного объекта. И если объект класса перестает использоваться, то при очистке стека ссылка на участок памяти также очищается, однако это не приводит к немедленной очистке самого участка памяти в куче.

Механизм сборки мусора в .NET

Сборщик мусора (garbage collector) очищает участок памяти в куче если «видит», что на него больше нет ссылок.

Чтобы понять, каким образом сборщик мусора «видит», когда объект уже не нужен, необходимо знать, что собой представляют корневые элементы приложения.

Корневым элементом называется ячейка в памяти, в которой содержится ссылка на размещающийся в куче объект.

Корневыми могут называться следующие элементы:

- Ссылки на глобальные объекты.
- Ссылки на любые статические объекты или статические поля.
- Ссылки на локальные объекты.
- Ссылки на передаваемые методу параметры объекта.
- Ссылки на объект, ожидающий финализации.
- Любые регистры центрального процессора, которые ссылаются на объект.

Механизм сборки мусора в .NET

Сборка мусора возникает при выполнении одного из следующих условий:

- Недостаточно физической памяти в системе.
- Объем памяти, используемой объектами, выделенными в управляемой куче, превышает допустимый порог.
- Вызывается метод **GC.Collect**. Практически во всех случаях вызов этого метода не потребуется, так как сборщик мусора работает непрерывно.

Во время сборки мусора строится граф используемых объектов, происходит исследование объектов в куче, с целью определения являются ли они достижимыми. Отправными точками в построении графа являются корневые объекты.

Если корневые объекты ссылаются на конкретные объекты в памяти, то они считаются живыми, все остальные - мусор, который надо удалить.

В конце происходит дефрагментация кучи – используемые объекты перераспределяются так, чтобы занимаемая ими память составляла единый блок в начале кучи.

Поколения объектов

Сборщик мусора не проверяет буквально каждый находящийся в куче объект, так как это большие затраты времени.

Для оптимизации процесса каждый объект в куче относится к определённому «поколению».

Смысл в применении поколений выглядит довольно просто:
Чем дольше объект находится в куче, тем выше вероятность того, что он там будет оставаться.

- **Поколение 0** – идентифицируется новый только что размещённый объект.
- **Поколение 1** – идентифицирует объект, который уже «пережил» один процесс сборки мусора.
- **Поколение 2** – идентифицирует объект, который пережил более одного прогона сбора мусора.

Если сборщик освобождает недостаточно памяти, перед вбрасыванием исключения **OutOfMemoryException** он выполняет полную сборку мусора.

Работа со сборщиком мусора из кода

Сборщик мусора в библиотеке классов .NET представлен классом **System.GC**.

Через статические методы и свойства данный класс позволяет обращаться к сборщику мусора.

int MaxGeneration { get; } – максимальный номер поколений объектов.

void Collect() – принудительный вызов сборщика мусора для всех поколений.

void Collect(int generation) – принудительный вызов сборщика мусора для поколений от **0** до **generation**.

GetGeneration(object obj) – номер поколения, где находится объект **obj**.

GetTotalMemory – общий объем памяти занимаемый кучей.

KeepAlive(object obj) – предотвращает удаление объекта при сборке мусора, даже если на него нет ссылок.

ReRegisterForFinalize(object obj) – добавляет объект в **FinalizationQueue**.

SuppressFinalize (object obj) – удаляет объект в **FinalizationQueue**.

WaitForPendingFinalizers – останавливает текущий поток до завершения выполнения методов **Finalize** для всех объектов из **F-reachable Queue**.

Утечка памяти

Утечка памяти – процесс неконтролируемого уменьшения объёма свободной памяти, связанный с ошибками в работающих программах, вовремя не освобождающих ненужные участки памяти, или с ошибками системных служб контроля памяти.

Существует много видов утечек памяти, но в целом этот термин относится к какому-то ресурсу, который больше не используется, но все еще занимает память. Если у вас их много, ваше приложение занимает много памяти, и в конечном итоге вы ее исчерпываете.

В C#, это некоторые общие утечки памяти:

- Не выполнена отписка от событий. Любой прослушиватель событий, созданный с помощью анонимного метода или выражения **lambda**, ссылающегося на внешний объект, сохранит эти объекты живыми.
- Не закрыты соединения с базой данные, если они не используются. Не забудьте вызвать **Dispose()** для всех объектов **IDisposable**. Используйте оператор **using** .
- Использование функций при помощи **Platform Invoke (P/Invoke)**, которые выделяют память но не освобождают.

Финализатор и уничтожение объекта

Финализатор (деструктор) – особый метод, который вызывается при уничтожении объекта. Деструктор следует реализовывать только у тех объектов, которым он действительно необходим, так как метод **Finalize** оказывает сильное влияние на производительность.

- При определении используется **~.**
- Имя деструктора соответствует имени класса.

```
IntPtr myHandle;

MyClass() {
   IntPtr myHandle = Marshal.AllocHGlobal(100);
}

~MyClass() {
    Marshal.FreeHGlobal(myHandle);
}
```

Жизненный цикл объектов

Выделить память для экземпляра типа, использующего ресурс.

- Оператор **new**.

Инициализировать выделенную память, чтобы установить начальное состояние ресурса.

- Открыть соединение с БД.
- Создать объект.

Использовать ресурс, обращаясь к членам экземпляра типа.

- Выполнять запросы с БД.
- Работать с объектом.

Разрушить состояние ресурса, чтобы выполнить очистку.

- Закрыть соединение с БД.
- Удалить ссылки на объект.

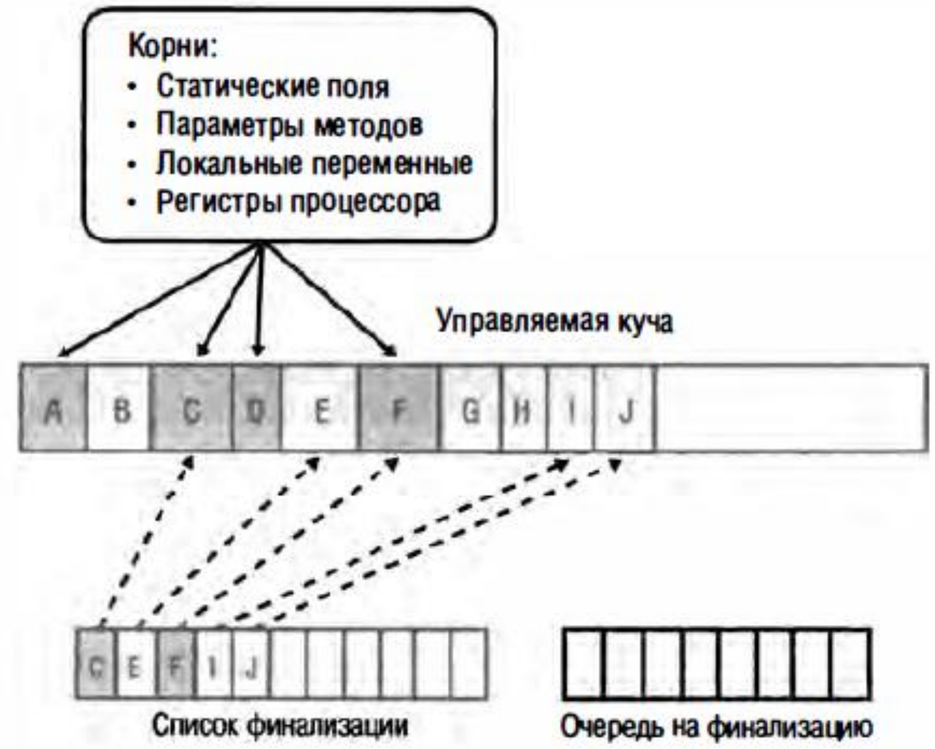
Освободить используемую память.

- Отвечает сборщик мусора.

Освобождение ресурсов

При компиляции деструкторы неявно переименовываются в **Finalize**. Данный метод вызывается перед уничтожением объекта.

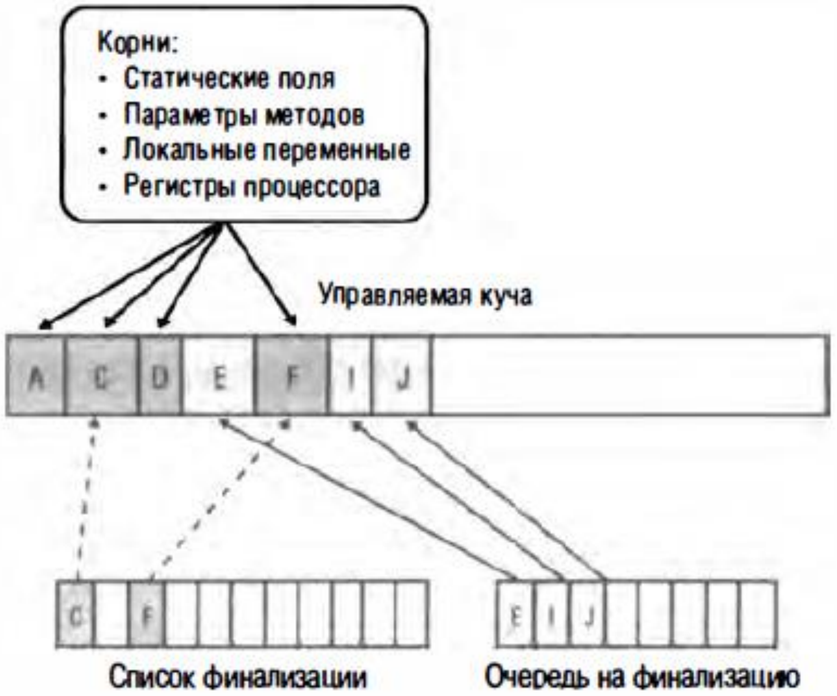
Указатели на объекты, имеющие метод **Finalize** помещаются в отдельную очередь **FinalizationQueue** перед вызовом конструктора типа.



Освобождение ресурсов

Отмечаются те объекты которые можно удалить.
Отмеченные для удаления объекты, которые были в **FinalizationQueue**

- Из общей памяти не удаляются
- Удаляется ссылка из **FinalizationQueue**
- Добавляется ссылка в **F-reachable Queue**

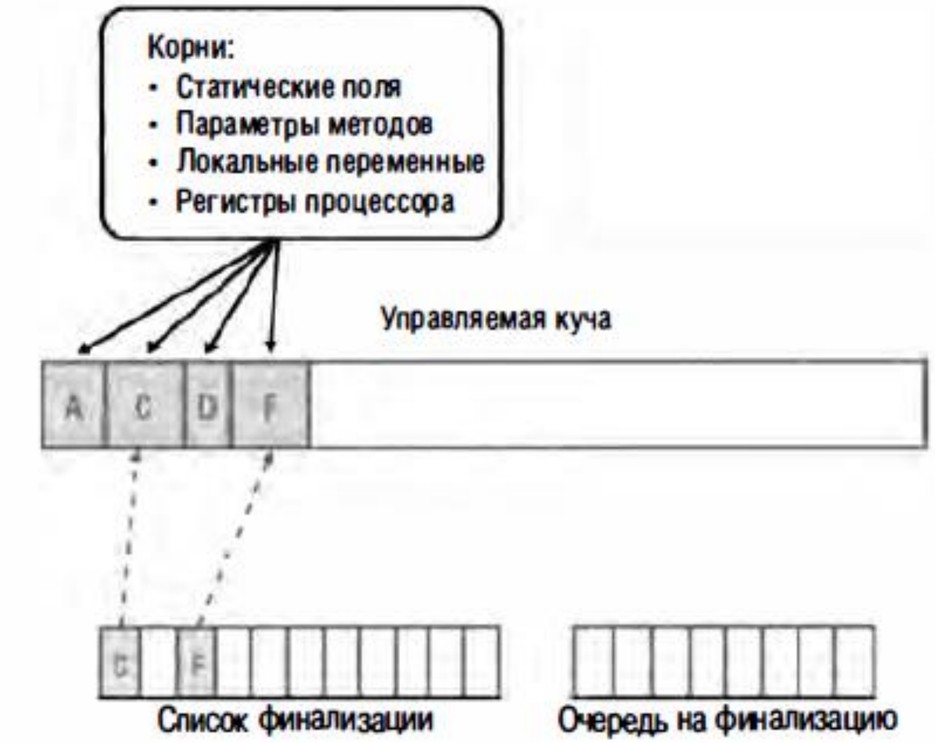


Освобождение ресурсов

Для каждого объекта **F-reachable Queue** вызывается метод **Finalize**.

Ссылки из **F-reachable** удаляются.

Объекты из кучи теперь отмечены как мусор и будут удалены при следующей сборке.



Паттерн Dispose

Объект может владеть управляемыми и неуправляемыми ресурсами, а также два способа освобождения ресурсов: с помощью метода **Dispose** и с помощью финализатора.

Паттерн **Dispose** по пунктам:

- Класс, содержащий управляемые или неуправляемые ресурсы реализует интерфейс **IDisposable.**
- Класс содержит метод **Dispose(bool disposing)**, который освобождает ресурсы, параметр **disposing** говорит о том, вызывается ли этот метод из метода **Dispose()** или из финализатора.
- Метод Dispose реализуется следующим образом: вначале вызывается метод **Dispose(true)**, а затем вызов метода **GC.SuppressFinalize()**, который предотвращает вызов финализатора.
- Метод **Dispose(bool disposing)** содержит две части: если этот метод вызван из метода **Dispose** - то мы освобождаем управляемые и неуправляемые ресурсы и если метод вызван из финализатора во время сборки мусора, то мы освобождаем только неуправляемые ресурсы.
- Класс может содержать финализатор и вызывать из него **Dispose(false)**.

Инструкция using

Разрабатывайте классы оболочки – для работы с неуправляемыми ресурсами.

При использовании классов работающих с неуправляемыми ресурсами (дескриптор файлов, сетевое подключение, и т.п.) вызывайте метод **Dispose** при завершении использования.

Используйте **using** при работе, он предоставляет удобный синтаксис, обеспечивающий правильное использование объектов **IDisposable**. Оператор **using** гарантирует вызов метода **Dispose**, даже если при вызове методов в объекте происходит исключение.

```
using (var stream = new FileStream("file.txt", FileMode.Open))
{
   byte[] buffer = new byte[stream.Length];
   await stream.ReadAsync(buffer, 0, (int)stream.Length);
}
```

Задание

Открыть задание из занятия по работе с файловой системой и переписать все, используя Dispose подход.