Внутренняя реализация

# Управляемая куча.

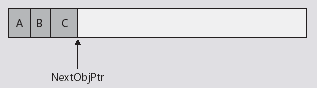
При инициализации процесса CLR резервирует единый сегмент адресного пространства. Это и есть управляемая куча. По мере необходимости сегмент памяти увеличивается, пока адресное пространство процесса не станет полным. Отмечу, что 64-битные приложения имеют значительно большее адресное пространство.

Управляемая куча также включает в себя указатель, назовем его NextObjPtr, который на участок кучи, где будет выделена память для нового объекта. Изначально он указывает на базовый адрес кучи.

Создает объект в куче команда IL-языка newobj. В эту команду преобразуется оператор new языка С#. newobj выполняет свою операции в 4 шага:

1. Вычисляет количество байт, необходимых для объекта.
2. Добавляет байты, необходимые для дополнительных данных. Сюда включается ссылка на объект типа, а также индекс синхронизации. В 32-битных приложениях это дополнительные 8 байт, 64-битных – 16 байт.
3. Проверяет наличие свободного пространства.
4. Если памяти хватает, то объект занимает свое пространство, NextObjPtr теперь указывает на память, сразу за созданным объектом.

Таким образом, управляемая память представляет собой нефрагментированный участок памяти , где каждый новый объект занимает участок в памяти сразу за предыдущим:



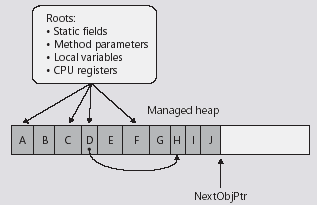
Для сравнения опишу, как работает виртуальная куча в C. Она представляет собой связный список выделенных объектов. При создании нового объекта в этом списке ищется достаточное место для нового объекта. И когда находится, объект «втискивается» между 2-мя. Когда объект удаляется из памяти (вручную – с помощью delete), объект удаляется из этого списка. Таким образом, сравнивая 2 кучи, мы видим, что, во-первых, выделение памяти для нового объекта в .NET происходит быстрее, поскольку нет нужды в линейном поиске свободного места. Во-вторых, объекты, созданные в одном коротком промежутке времени, всегда помещаются рядом в памяти. Поскольку, как правило, объекты, созданные примерно в одно и то же время, часто используются вместе, то близкое расположение в памяти уменьшает время обращения к RAM. Увеличивается и шанс близким объектам попасть в кэш CPU.

# Сборщик мусора.

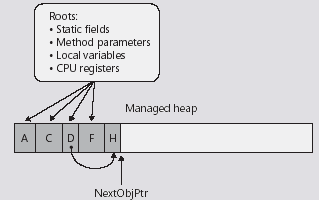
В случае, когда для нового объекта не хватает памяти, запускается сборщик мусора.

Изначально, сборщик считает все объекты в куче мусором. Он начинает поиск достижимых объектов, т.е. тех, которые могут использоваться в приложении. Для этого сначала необходимо найти все «корни» – т.е. стартовые объекты, от которых нужно искать достижимые объекты. Сюда входят статические элементы, локальные переменные и параметры выполняемой функции, а также всех остальных функций стека вызова.

Первым делом сборщик идет по стеку потока, считая, что все переменные в нем – корни. Если объект – Reference type, то ему выставляется специальный бит в индексе синхронизации. Обнаружив все корни, сборщик начинает их рекурсивный обход, переходя по всем ссылкам каждого очередного объекта. Каждый новый объект получает все тот же бит доступности. Обход заканчивается, если очередной объект уже имеет выставленным бит доступности.



Получив таким образом набор доступных объектов, сборщик начинает следующую фазу «уплотнение». Сборщик «сдвигает» на место недостижимых объектов оставшиеся объекты.



Такой «сдвиг» делает все указатели неправильными, поэтому сборщику нужно выставить правильные значения все объектам. Он снова начинает рекурсивно идти по объектам, начиная с корней.

# Поколения.

Использование поколений – способ оптимизировать сборку мусора. Он делает следующие предположения, опираясь на которые делает сборку мусора быстрее:

1. Более новые объекты большей частью имеют меньшую «продолжительность жизни».
2. Старые объекты, как правило, продолжают использоваться более продолжительное время.
3. «Уплотнение» на части кучи происходит быстрее, чем на всей куче.

Сборщик мусора разбивает всю кучу на поколения. Каждому поколению выделяется определенный размер памяти. Поколением 0 называется часть кучи, содержащая ни разу не проходившие через сборку мусора.

После сборки мусора все пережившие сборку мусора объекты помещаются в поколение 1. При следующей сборке мусора проверяется, достигло поколение 1 максимального размера, если нет, то сборка будет происходить только на поколении 0. Построение дерева достижимых объектов будет прерываться не только на уже помеченных объектах, но и на объектах их поколения 1. Чтобы отследить, что на объекты из поколения 0 стали ссылаться объекты из поколения 1 во время работы приложения, JIT-компилятор добавляет «барьерный» метод при изменениях в полях объекта. Этот метод проверяет, проверяет, принадлежит ли объект к поколению 1 или 2, и, если принадлежит, добавляет объект в таблицу «card table». Только объекты, попавшие в эту таблицу, будут проверяться на наличие ссылок на новые объекты.

Если при сборке мусора поколение 1 также достигнет своего максимального значения, то сборка мусора будет проводиться и на поколении 1, и на поколении 0. Объекты из 1 поколения, пережившие сборку мусора перейдут в поколение 2, а из 0-го – в поколение 1.

Если при сборке все 3 поколения достигли своего максимума, то будет проходить полная сборка мусора, но в 3-е поколение никто переходить не будет.

CLR определяет размер поколения 0 в 256KB, 1 – 2MB, 2 – 10MB, однако эти величины могут меняться самим сборщиком вследствие различных условий.

# Освобождение ресурсов.

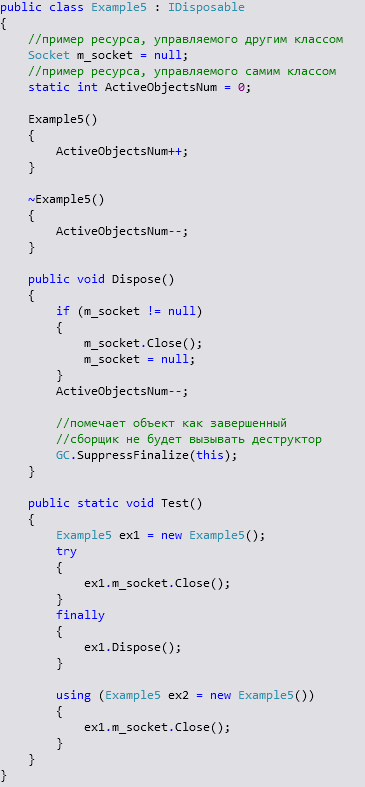
Кроме памяти объекты могут также использовать другие ресурсы, которые необходимо освобождать, когда объект перестает использоваться. В .NET введена специальная модель освобождения таких ресурсов.

Первый способ освобождения ресурсов – использование «деструкторов». Несмотря на синтаксис, похожий на язык С++, деструктор не освобождает память объекта и не может вызываться вручную. Деструктор вызывается, когда сборщик мусора пометил объект как мусор и готовится удалить его из памяти. Деструктор должен освободить только те ресурсы, которыми управляет напрямую. Если у объекта есть члены, которые сами управляют своими ресурсами, то нет необходимости в освобождении ресурсов у этих объектов, они должны сделать это сами. Более того, если оба эти объекта стали мусором, то один из объектов уже может быть удален из памяти и обращение к нему в деструкторе уже невозможно.

Использование деструкторов для освобождения ресурсов, очевидно, является не самым оптимальным решением, поскольку объект перестал использоваться задолго до того, как сработает сборщик мусора, и вызовется деструктор, а связанные с объектом ресурсы всё это время будут использоваться. Так что наиболее верным способом является ручное освобождение ресурсов, а использование деструктора скорее является защитным механизмом, если программист забыл выполнить освобождение ресурсов вручную.

Для освобождения ресурсов вручную используется интерфейс *IDisposable.* Наследуя свой класс от этого интерфейса, вы заявляете, что используете в классе ресурсы, которые необходимо освобождать. Причем, в отличие от деструктора, вручную вы также должны освобождать и те ресурсы, которые управляются объектами, являющимися членами вашего класса.

Интерфейс *IDisposable* cодержит только 1 метод – *Dispose().* Вызывая этот метод пользователь объект говорит, что этот объект больше не нужен и не будет использоваться, и потому необходимо освободить все ресурсы, которыми он оперирует, включая те ресурсы, которые управляются посредством объектов – членов вашего класса.

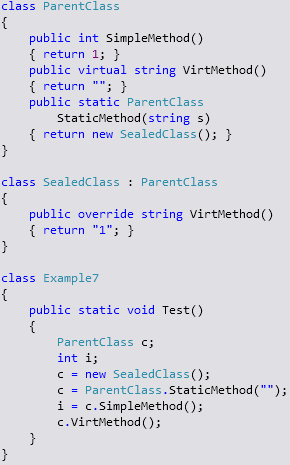


Необходимо следить за тем, чтобы *Dispose* выполнился, в частности, иметь в виду исключения, которые могут прервать код, поэтому рекомендуется помещать вызов метода в блок *finally*. Для упрощения конструкции в С# введен оператор using, который добавляет блок *finally* с вызовом *Dispose.*

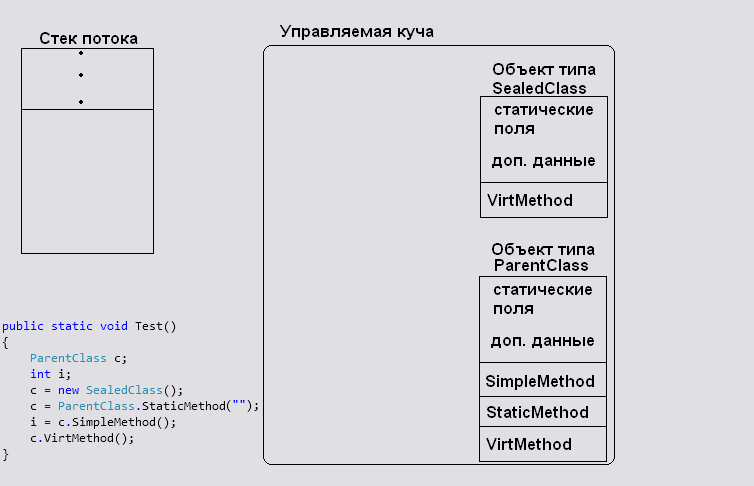
# Внутреннее взаимодействие объектов.

Все данные хранятся либо в стеках потоков, либо в управляемой куче. Причем на данные из кучи ссылаются переменные из стеков. Ссылаться на данные из стеков, в отличие от С, нельзя.

Рассмотрим на примере следующего кода, как происходит взаимодействие объектов:

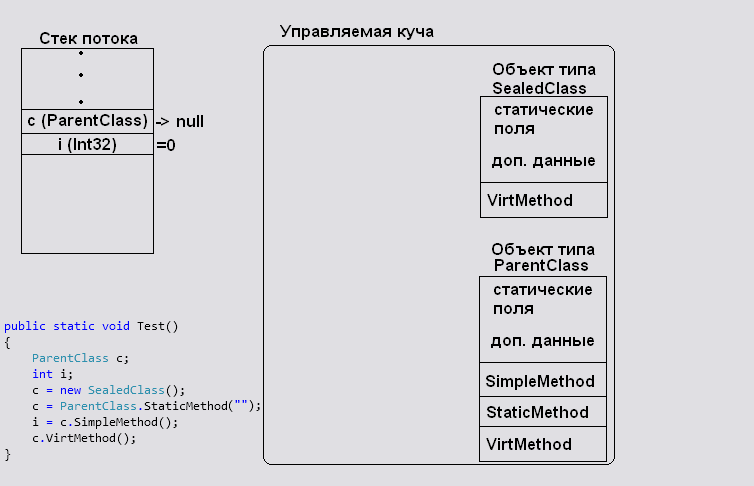


Рассмотрим выполнение функции *Test* пошагово. Если вызов функции происходит в первый раз, то JIT-компилятору необходимо скомпилировать IL-код, содержащийся в этой функции.

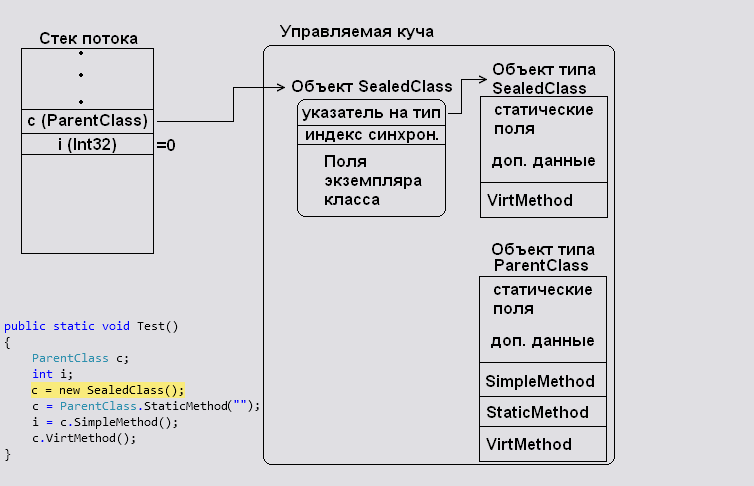


Во время компиляции JIT-компилятор выясняет, какие типы использованы в функции. Далее, используя данные метаданных, он создает в куче объекты всех этих типов, если они уже не находятся там. Объекты типов содержат 3 типа данных:

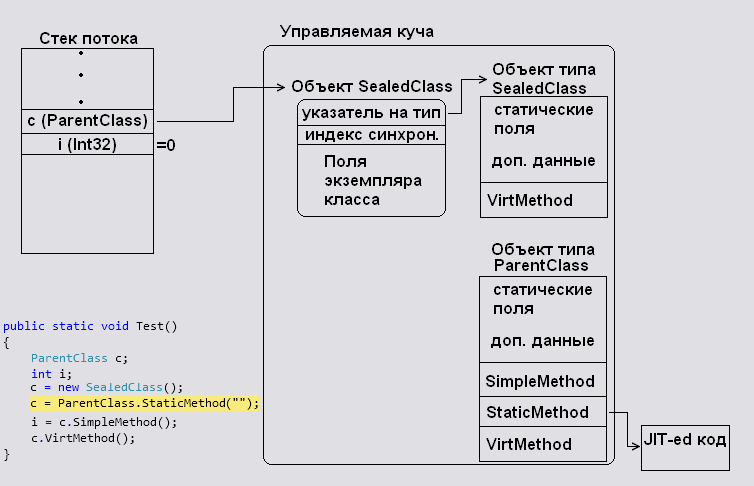
1. Данные статических полей
2. Ссылки на методы класса. До первого вызова соответствующего метода эта ссылка содержит null, после вызова и, соответственно, компиляции метода ссылка указывает от откомпилированный с помощью JIT код.
3. Дополнительные данные:
   1. Ссылка на объект типа парента. Необходима, например, при поиске функции, объявленной у парента.
   2. Утверждается, что объекты типа также содержат (как и все остальные объекты в куче) поля ссылки на тип и индекс синхронизации, однако зачем они нужны объектам типа непонятно. Ссылка на себя явно излишня, а индекс синхронизации используется сборщиком. Объекты типа явно являются корнями, и им тоже этот индекс не нужен.



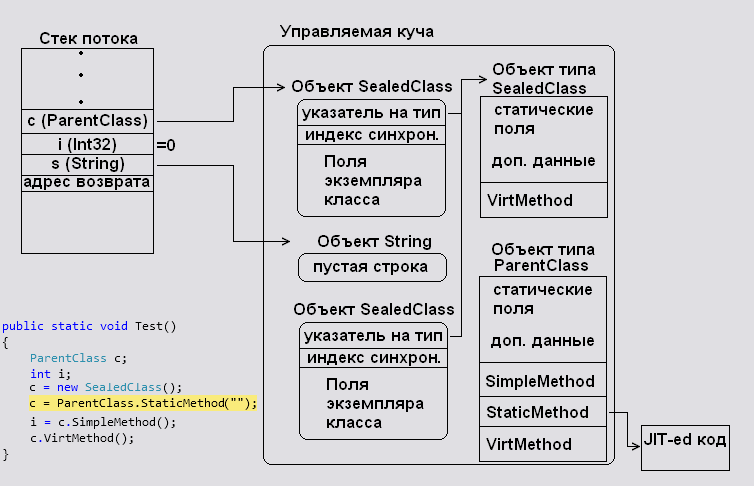
После инициализации объектов типа инициализируются все локальные переменные в функции. Они помещаются в стек потока, при этом все байты данных обнуляются. Однако попытка использовать переменные до присваивания им значений приведет к ошибке компиляции.



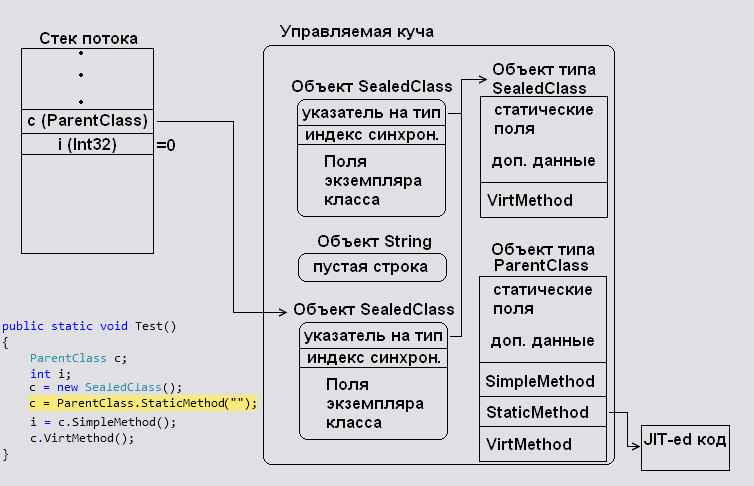
Первым шагом в функции станет создание экземпляра класса *SealedClass*. В куче создается объект, содержащий все поля, объявленные в типе *SealedClass*, а также всех его родительских классах. CLR добавляет ссылку на объект типа, а также индекс синхронизации. Ссылка на созданный объект присваивается переменной *c.*



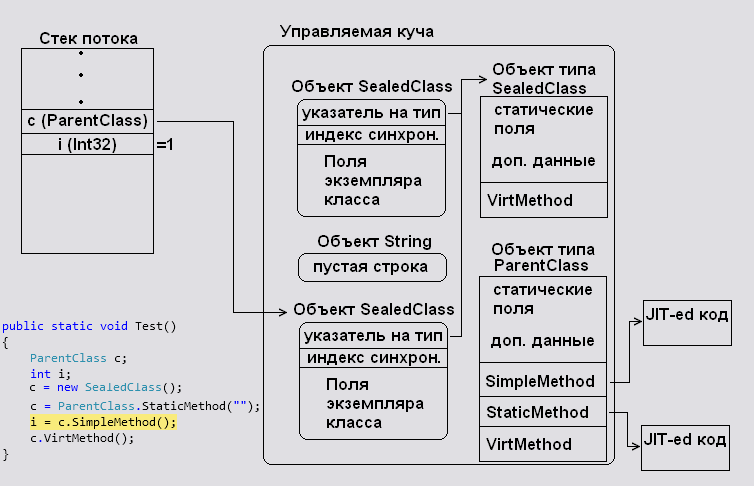
Следующий шаг – выполнение статической функции. Сначала CLR переходит к объекту типа *ParentClass*, в нем ищет метод *StaticMethod.* Поскольку он еще ни разу не вызывался, то ссылка на метод равна нулу. JIT компилирует метод и выставляет ссылку на него в объект типа. После этого вызывается сам метод.



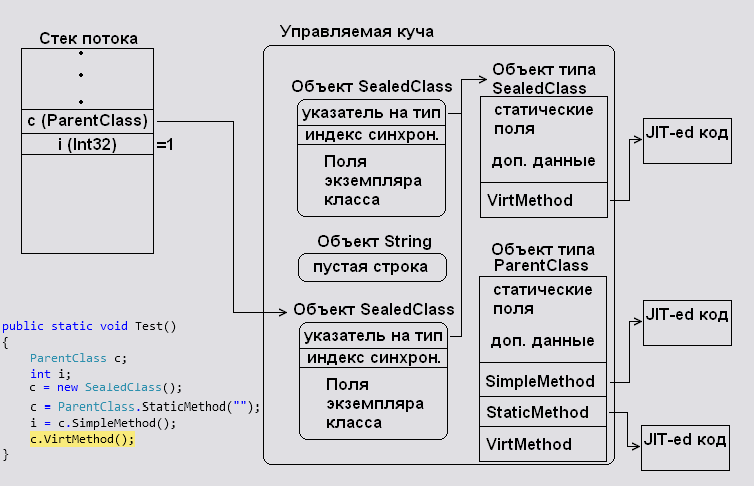
Как и в случае с функцией *Test*, CLR сначала ищет все типы, используемые в функции. Так как новых типов нет, то новых объектов типов не создается. Далее в стеке инициализируется параметр функции – *s*. Он ссылается на новый объект в куче – строку. Этот объект имеет тот же набор данных, кроме полей класса. Также в стек заносится *адрес возврата*, определяющий куда должен вернуться CLR после окончания выполнения функции. В самой функции создается еще один экземпляр класса *SealedClass*. Ссылка на него возвращается из функции.



После выхода из функции стек очищается от объектов функции, а переменная c теперь указывает на новый объект *SealedClass*. Обратите внимание, что старый объект *SealedClass*, а также объект строки стали мусором. Они будут очищены при следующей сборке мусора.



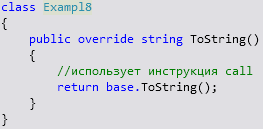
Далее вызывается невиртуальный нестатичный метод. В этом случае CLR ищет объект типа, к которому принадлежит переменная. Если у этому объекта типа нет необходимого метода, то CLR идет по списку парентов типа. Метод еще не вызывался, поэтому его необходимо скомпилировать. После компиляции ссылка на метод присваивается соответствующему методу в объекте типа, а после этого вызывается сам метод. Как и в предыдущей функции снова в стеке инициализируется новые переменные, а также адрес возврата. После исполнения метода стек от них очищается, а переменной *i* присваивается результат функции.



Наконец, последний оператор в функции *Test*. Это виртуальный метод. Для него CLR ищет объект типа, используя ссылку «указатель на тип» в самом объекте. Если там искомого метода нет, то CLR снова ищет его среди родителей. Снова проверяется, откомпилирован ли этот метод, и если нет, то компилируется, а ссылка на него помещается в соответствующий метод объекта типа.

# Вызов виртуальных и невиртуальных методов.

Описанный выше механизм вызова виртуальных и невиртуальных методов на самом деле несколько сложнее. В IL-языке есть 2 инструкции *call* и *callvirt*. Описанный выше механизм как раз действует для этих 2 инструкций, однако компилятор C# не всегда вызов виртуального метода переводит в инструкцию *callvirt*, а невиртуального – в *call*. Для этого есть несколько причин. *Callvirt*, кроме собственно вызова метода по схеме вызова виртуальных функций, делает проверку, что переменная, от которой вызывается метод не null. Именно ради этой проверки компилятор C# использует эту инструкцию для невиртуальных методов. Дело в том, что компилятор при этом еще имеет в виду JIT-компилятор, который на ходу оптимизирует вызов функции с «виртуальной» схемы на «невиртуальную», поскольку вызываемая функция невиртуальна. Причина же использования инструкции *call* для виртуальных методов приведена в примере:



Если бы была использована инструкция *callvirt*, произошло бы зацикливание – каждый раз метод класса *Exampl8* вызывал бы сам себя.

Другой пример использования инструкции *call* для виртуального метода – когда переменная принадлежит классу с модификатором *sealed*, т.е. у класса не может быть потомков.