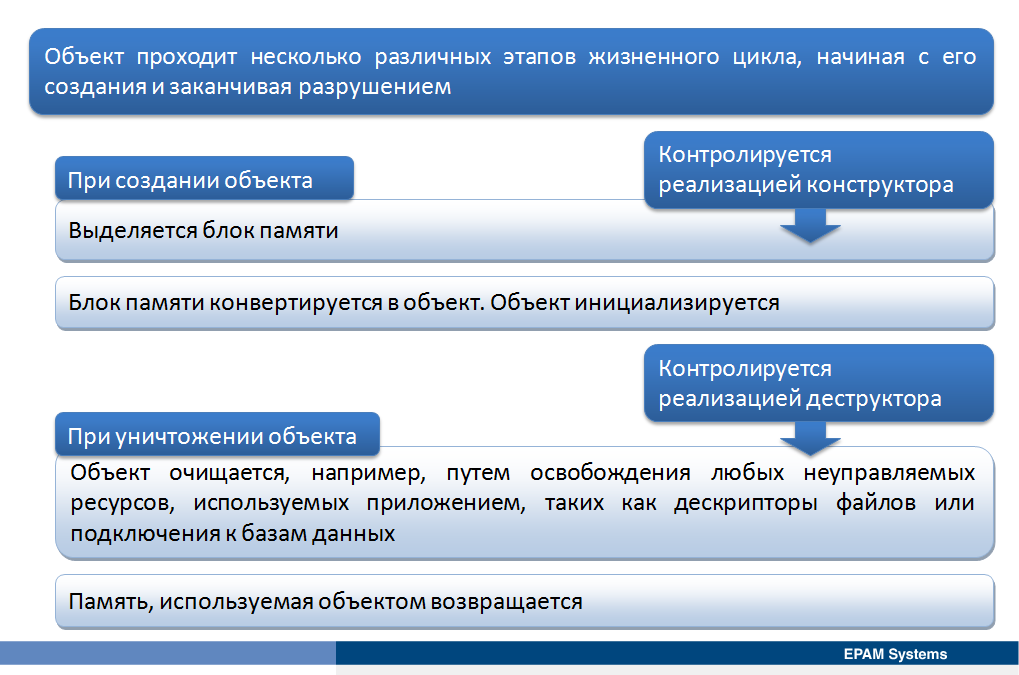
**Сборка мусора, взаимодействие с файловой системой**

**Жизненный цикл объектов.**



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.

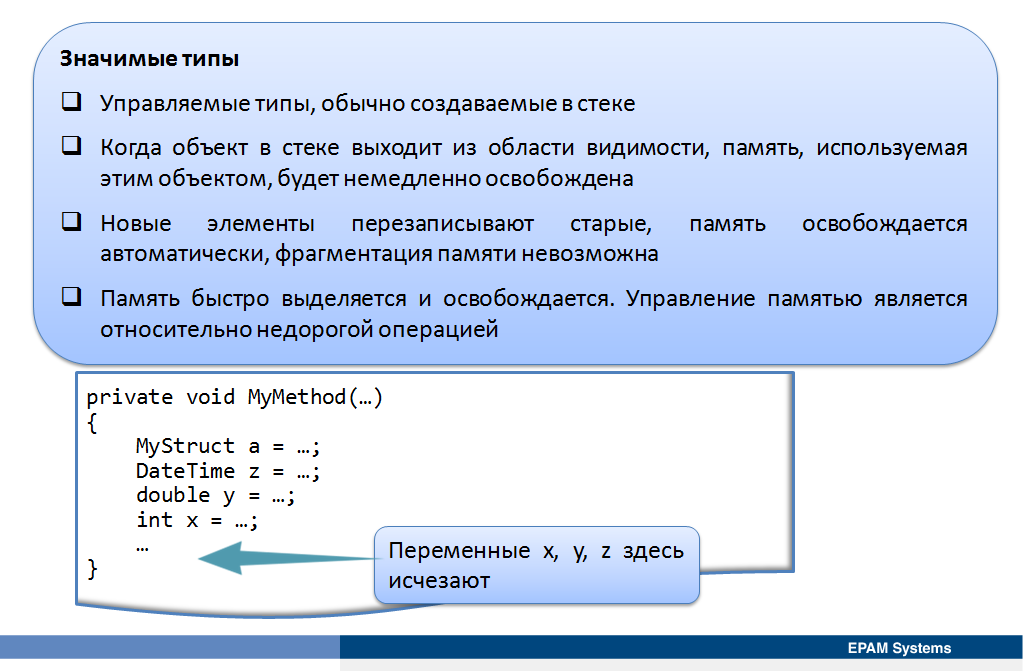
Когда работа с объектом завершена, он может быть уничтожен. Для возвращения любых ресурсов, используемых этим объектом, используется уничтожение (деструкция). Подобно созданию уничтожение это двухэтапный процесс:

* 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.

**Понятие сборщика мусора.**



.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 создании 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.

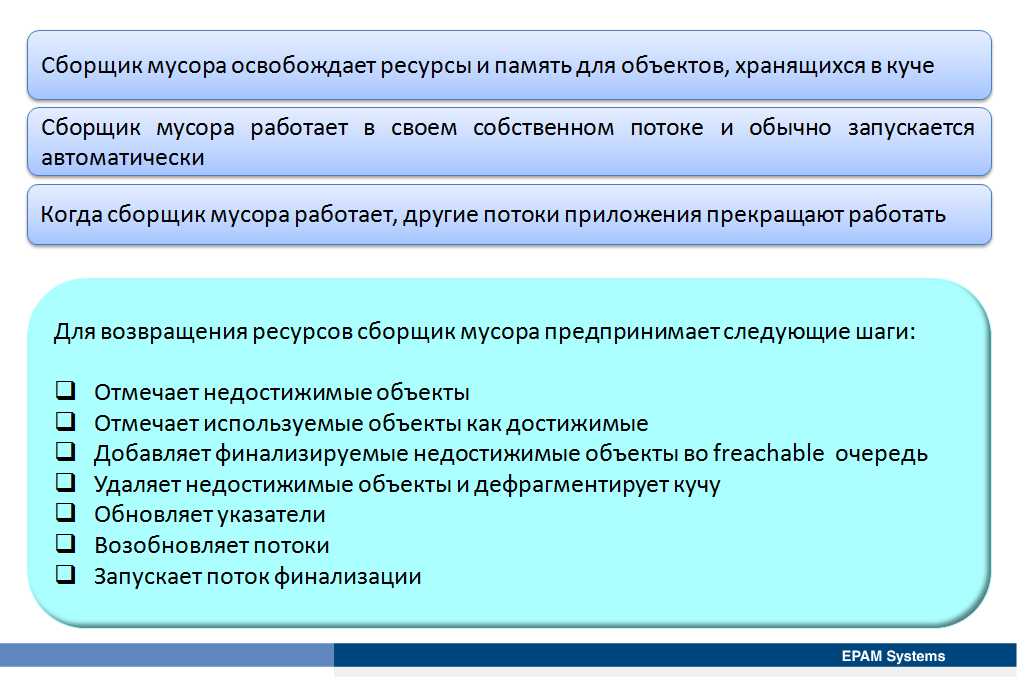
Важной функцией сборщика мусора .NET Framework является наблюдение за объектом в куче и определение, когда последняя ссылка на этот объект исчезнет, тогда объект может быть безопасно уничтожен. Определение момента, когда объект не имеет ссылки, может быть трудоемкой и дорогостоящей операцией, поэтому 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 результирующее 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.



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

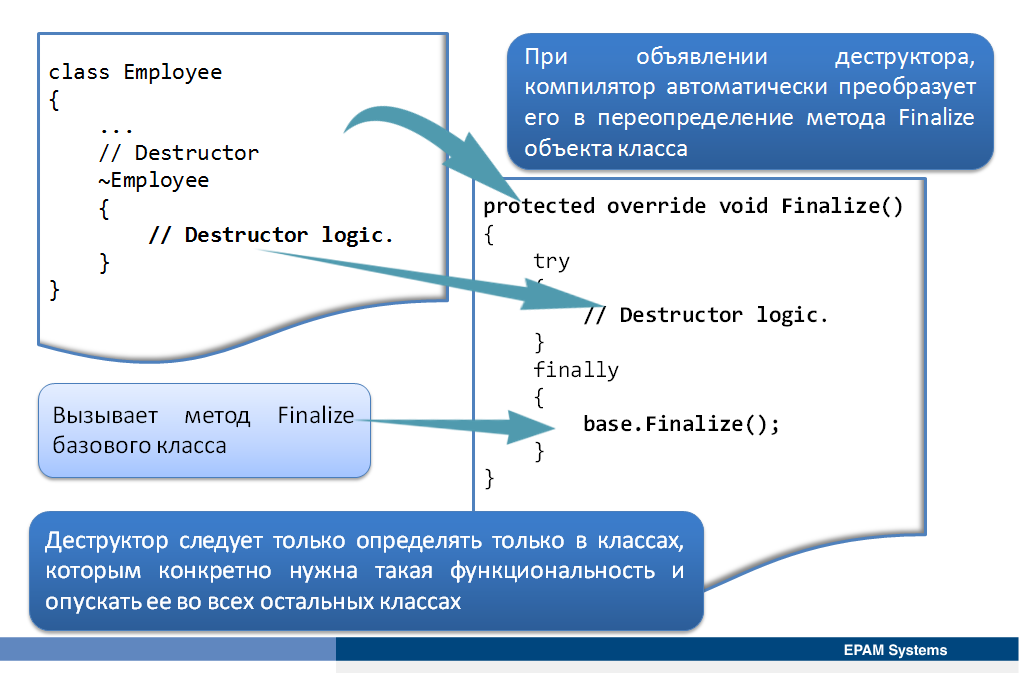
NextObjPtr

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.

**Деструктор и метод Finalize.**



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

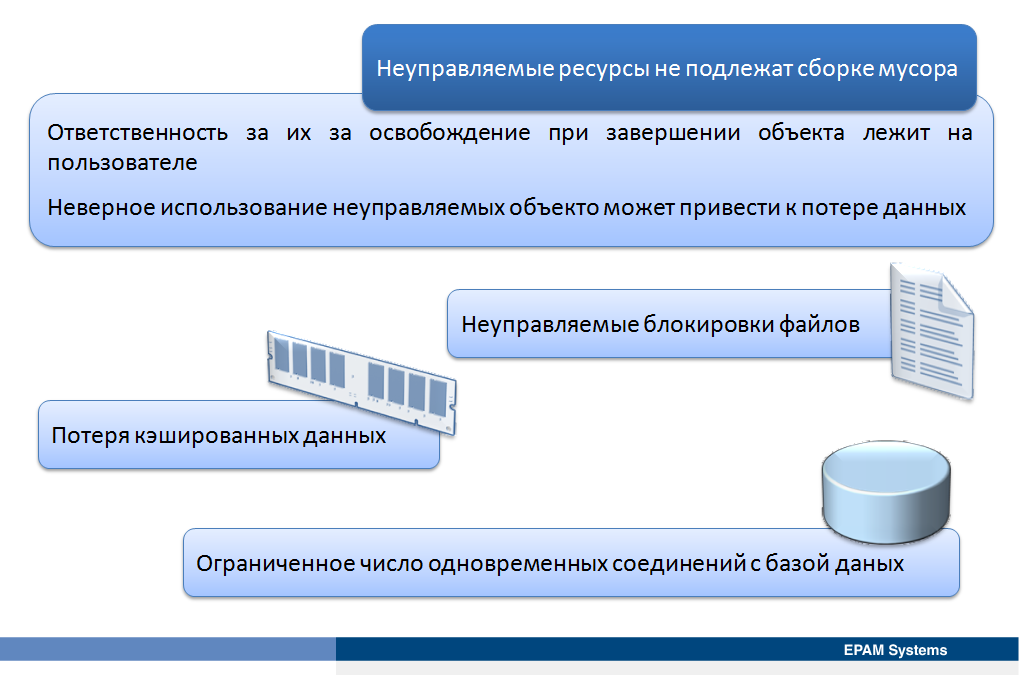
Статические

Локальные

Регистры процессора

Рис. 6.

*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.

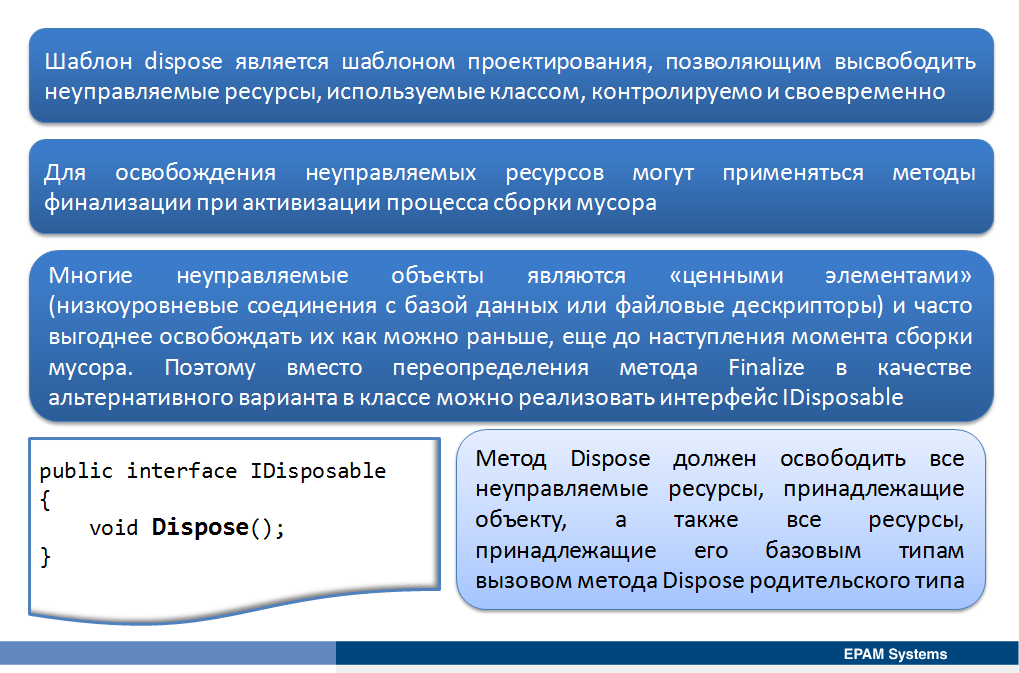
Библиотека классов .NET Framework, обеспечивает, например, класс TextWriter, используемый для открытия файла локальной файловой системы и записи текста в файл. Класс TextWriter действует как управляемая оболочка вокруг текстовых файлов, являющихся неуправляемыми ресурсами, контролируемыми операционной 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 высвободить 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.

**Метод Dispose и интерфейс IDisposable.**



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()

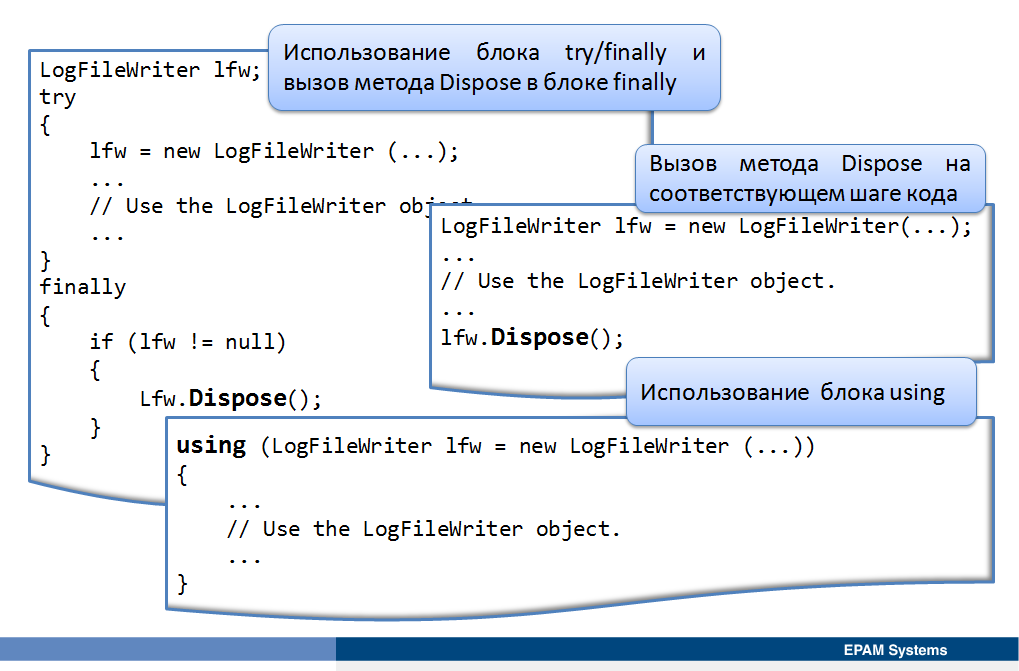
{

Dispose(true);

GC.SuppressFinalize(this);

}

*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)

{

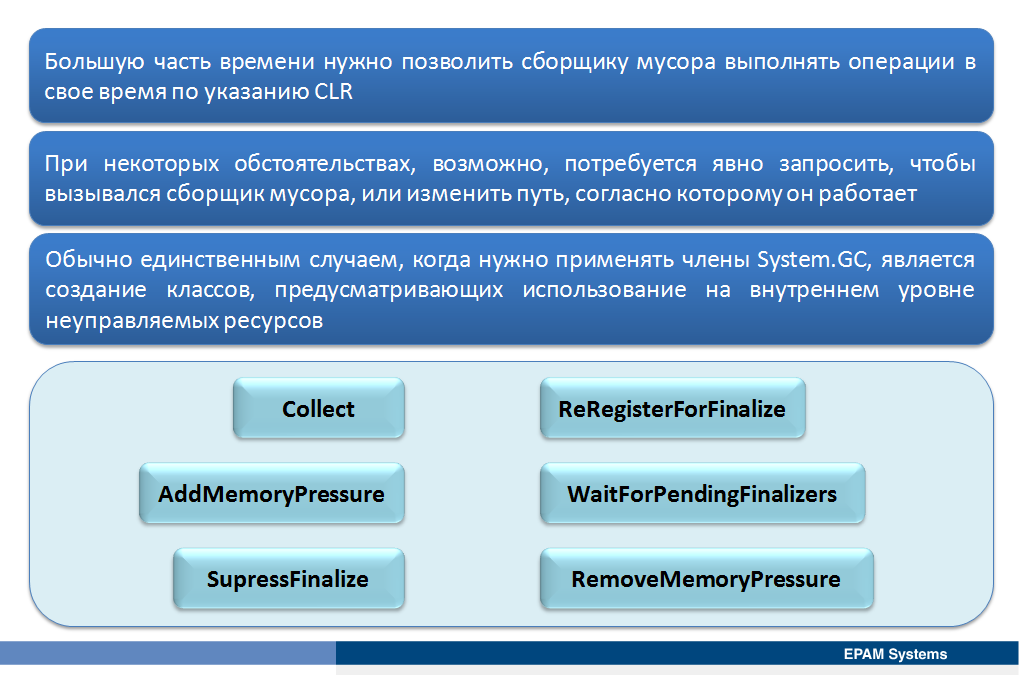
lfw.Dispose();

}

}

}

**Класс System.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, следует 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); |

Понятие поколений при сборке мусора.

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.

**Модель потоков в C#. Пространство System.IO.**



При работе с данными, будь то данные хранимые в файле файловой системы или на веб-сервере, доступном через соединение HTTP, данные иногда становится слишком большими, чтобы загружать их в память и передавать одной атомарной операцией. Например, можно представить попытку загрузить в память видео файл в 100-гигабайт в одну операцию. Мало того, что такая операция занимает много времени, она также потребляют большой объем памяти.

Для решения обозначенной проблемы .NET Framework позволяет использовать потоки. Поток представляет собой последовательность байтов, которые могут поступать из файла файловой системы, сети связи или памяти. Потоки позволяют считывать или записывать данные в источник данных посредством небольших управляемых пакетов данных. Как правило, потоки обеспечивают следующие операции:

* Чтение из потока – это перенос информации из потока в структуру данных, такую как массив байтов.
* Запись в поток – это передача данных из структуры данных в поток.
* Поиск в потоке – это выяснение и изменение текущей позиции внутри потока. Возможность поиска зависит от вида резервного хранилища потока. Например, в сетевых потоках отсутствует унифицированное представление текущего положения, поэтому обычно они не поддерживают поиск.

.NET Framework предоставляет несколько потоковых классов, позволяющих работать с различными данными и источниками данных. Для выбора какого-либо из них, необходимо учитывать следующее:

* Какой тип данных читается или записывается, например, двоичный или буквенно-цифровой.
* Где хранятся данные, например, в локальной файловой системе, в памяти или на веб-сервере в сети.

Библиотека классов .NET Framework предоставляет несколько классов из пространства имен System.IO, которые можно использовать для чтения и записи файлов с помощью потоков. На самом высоком уровне абстракции (Рис. 1.), класс Stream определяет общие функциональные возможности, которые обеспечивают все потоки; он представляет универсальное представление последовательности байтов вместе с операциями и свойствами, которые обеспечивают все потоки. Класс Stream и его производные классы предоставляют универсальное представление различных типов ввода и вывода, изолируя программиста от отдельных сведений операционной системы и базовых устройств.

Рис. 1.

Внутренне, объект Stream поддерживает указатель, который ссылается на текущее местоположение в источнике данных. Когда впервые строится объект Stream для источника данных, этот указатель позиционируется перед первым байтом. Когда происходит чтение или запись данных, класс Stream продвигает этот указатель к концу данных, которые читаются или записываются. Класс Stream нельзя использовать напрямую. Вместо этого, нужно реализовать специализации этого класса, которые оптимизированы для выполнения потокового ввода/вывода для конкретных типов источников данных. Таким образом, в каждом резервном хранилище (резервное хранилище – устройство хранения информации, например диск или память) используется собственный поток как реализация класса [Stream](http://msdn.microsoft.com/ru-ru/library/system.io.stream(v=VS.90).aspx). Например, класс FileStream реализует поток, который использует файл на диске в качестве источника данных, класс MemoryStream реализует поток, который использует блок памяти в качестве источника данных. Потоки, связанные с резервными хранилищами, называются базовыми. В конструкторах базовых потоков есть параметры, необходимые для присоединения потока к резервному хранилищу. Например, класс [FileStream](http://msdn.microsoft.com/ru-ru/library/system.io.filestream(v=VS.90).aspx) имеет конструктор, задающий путь, который определяет порядок совместного использования файла процессами и т. д.

Структура классов [System.IO](http://msdn.microsoft.com/ru-ru/library/system.io(v=VS.90).aspx) предоставляет упрощенное составление потоков. Базовые потоки могут быть присоединены к одному или нескольким сквозным потокам, которые обеспечивают требуемую функциональность. Модуль чтения или записи может быть присоединен к концу цепочки, что обеспечит легкое считывание или запись предпочитаемых типов.

Пространство имен System.IO в .NET — это область библиотек базовых классов, посвященная службам файлового ввода-вывода, а также ввода-вывода из памяти. Подобно любому пространству имен, в System.IO определен набор классов, интерфейсов, перечислений, структур и делегатов, большинство из которых находятся в mscorlib.dll. В дополнение к типам, содержащимся внутри mscorlib.dll, в сборке System.dll определены дополнительные члены пространства имен System.IO. Обратите внимание, что все проекты Visual Studio 2010 автоматически устанавливают ссылки на обе сборки.

Многие типы из пространства имен System.IO сосредоточены на программных манипуляциях физическими каталогами и файлами. Дополнительные типы предоставляют поддержку чтения и записи данных в строковые буферы, а также области памяти. Ниже кратко описаны основные (неабстрактные) классы, которые дают понятие о функциональности System.IO:

**BinaryReader, BinaryWriter**

Эти классы позволяют сохранять и извлекать элементарные типы данных (целочисленные, булевские, строковые и т.п.) в двоичном виде

**BufferedStream**

Этот класс предоставляет временное хранилище для потока байтов, которые могут затем быть перенесены в постоянные хранилища

**Directory, DirectoryInfo**

Эти классы используются для манипуляций структурой каталогов машины. Тип Directory представляет функциональность, используя статические члены. Тип DirectoryInfo обеспечивает аналогичную функциональность через действительную объектную ссылку

**DriveInfo**

Этот класс предоставляет детальную информацию относительно дисковых устройств, используемых данной машиной

**File, FileInfo**

Эти классы служат для манипуляций множеством файлов данной машины. Тип File представляет функциональность через статические члены. Тип FileInfo обеспечивает аналогичную функциональность через действительную объектную ссылку

**FileStream**

Этот класс обеспечивает произвольный доступ к файлу (т.е. возможности поиска) с данными, представленными в виде потока байт

**FileSystemWatcher**

Этот класс позволяет отслеживать модификации внешних файлов в определенном каталоге

**MemoryStream**

Этот класс обеспечивает произвольный доступ к данным, хранящимся в памяти, а не в физическом файле

**Path**

Этот класс выполняет операции над типами System.String, содержащими информацию о пути к файлу или каталогу в независимой от платформы манере

**StreamWriter, StreamReader**

Эти классы используются для хранения (и извлечения) текстовой информации из файла. Эти классы не поддерживают произвольного доступа к файлу

**StringWriter, StringReader**

Подобно классам StreamWriter/StreamReader, эти классы также работают с текстовой информацией. Однако лежащим в основе хранилищем является строковый буфер, а не физический файл

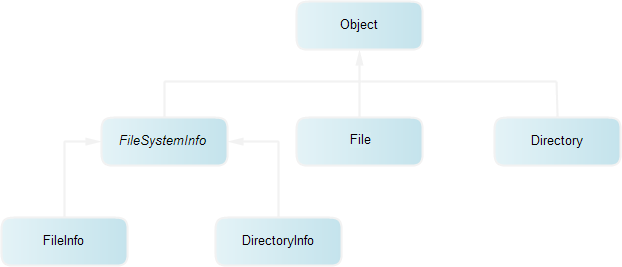
В дополнение к этим конкретным типам классов в System.IO определено несколько перечислений, а также набор абстрактных классов (т.е. Stream, TextReader и TextWriter), которые определяют разделяемый полиморфный интерфейс для всех наследников.

Как не трудно было заметить в приведенном выше списке, для представления файлов и папок используются по два класса. Какой из них применять — во многом зависит от того, сколько раз требуется получить доступ к данной папке или файлу.

* Классы Directory и File содержат только статические методы, а их экземпляры никогда не создаются. Для их использования нужно просто предоставить путь к интересующему объекту файловой системы при каждом вызове метода-члена. Если требуется выполнить только одну операцию в отношении папки или файла, то применение этих классов является наиболее эффективным подходом, поскольку позволяет избегать накладных расходов, связанных с созданием экземпляров.
* Классы DirectoryInfo и FileInfo реализуют практически те же общедоступные методы, что и Directory и File, а также некоторые общедоступные свойства и конструкторы, но поддерживают состояние, а их члены не являются статическими.

Сначала понадобится создать их экземпляр, который затем ассоциировать с определенной папкой или файлом. Это означает, что применение этих классов будет более эффективным подходом, если необходимо выполнять множество операций в отношении одного и того же объекта. Дело в том, что во время создания они считывают информацию об аутентификации и прочие сведения о соответствующем объекте файловой системы и больше не нуждаются в ее повторном чтении, сколько бы методов не вызывалось для данного объекта (экземпляра класса). В отличие от этого, классы, не поддерживающие состояние, нуждаются в проверке деталей, связанных с файлом или папкой, при каждом вызове какого-либо метода.

Обратите внимание на рисунке, что классы Directory и File непосредственно расширяют System.Object, в то время как DirectoryInfo и FileInfo наследуются от абстрактного класса **FileSystemInfo**:



Классы DirectoryInfo и FileInfo унаследовали значительную часть своего поведения от абстрактного базового класса FileSystemInfo. По большей части члены класса FileSystemInfo используются для получения общих характеристик (таких как время создания, различные атрибуты и т.д.) определенного файла или каталога. В таблице ниже перечислены некоторые основные свойства, представляющие интерес:

|  |  |
| --- | --- |
| Свойство | Назначение |
| **Attributes** | Получает или устанавливает ассоциированные с текущим файлом атрибуты, которые представлены перечислением FileAttributes (доступный только для чтения, зашифрованный, скрытый или сжатый) |
| **CreationTime** | Получает или устанавливает время создания текущего файла или каталога |
| **Exists** | Может использоваться для определения, существует ли данный файл или каталог |
| **Extension** | Извлекает расширение файла |
| **FullName** | Получает полный путь к файлу или каталогу |
| **LastAccessTime** | Получает или устанавливает время последнего доступа к текущему файлу или каталогу |
| **LastWriteTime** | Получает или устанавливает время последней записи в текущий файл или каталог |
| **Name** | Получает имя текущего файла или каталога |

В классе FileSystemInfo также определен метод Delete(). Этот метод реализуется производными типами для удаления файла или каталога с жесткого диска. Кроме того, метод Refresh() может быть вызван перед получением информации об атрибутах, чтобы обеспечить актуальность состояния статистики о текущем файле или каталоге.

**Класс Stream.**

Основным для потоков является класс **System.IO.Stream**. Он представляет байтовый поток и является базовым для всех остальных классов потоков. Кроме того, он является абстрактным классом, а это означает, что получить экземпляр объекта класса Stream нельзя.

В абстрактном классе System.IO.Stream определен набор членов, которые обеспечивают поддержку синхронного и асинхронного взаимодействия с хранилищем (например, файлом или областью памяти).

Концепция потока не ограничена файловым вводом-выводом. Точности ради следует отметить, что библиотеки .NET предоставляют потоковый доступ к сетям, областям памяти и прочим абстракциям, связанным с потоками.

Опять-таки, потомки класса Stream представляют данные, как низкоуровневые потоки байт, а непосредственная работа с низкоуровневыми потоками может оказаться довольно загадочной. Некоторые типы, унаследованные от Stream, поддерживают поиск (seeking), что означает возможность получения и изменения текущей позиции в потоке. Чтобы приблизиться к пониманию функциональности класса Stream, рассмотрим список основных его членов, приведенный ниже:

|  |  |
| --- | --- |
| Метод, свойство | Описание |
| **CanRead, CanWrite, CanSeek** | Определяют, поддерживает ли текущий поток чтение, поиск и/или запись |
| **Close()** | Закрывает текущий поток и освобождает все ресурсы (такие как сокеты и файловые дескрипторы), ассоциированные с текущим потоком. Внутренне этот метод является псевдонимом Dispose(). Поэтому закрытие потока функционально эквивалентно освобождению потока |
| **Flush()** | Обновляет лежащий в основе источник данных или репозиторий текущим состоянием буфера с последующей очисткой буфера. Если поток не реализует буфер, метод не делает ничего |
| **Length** | Возвращает длину потока в байтах |
| **Position** | Определяет текущую позицию в потоке |
| **Read(), ReadByte()** | Читает последовательность байт (или одиночный байт) из текущего потока и перемещает текущую позицию потока на количество прочитанных байтов |
| **Seek()** | Устанавливает позицию в текущем потоке |
| **SetLength()** | Устанавливает длину текущего потока |
| **Write(), WriteByte()** | Пишет последовательность байт (или одиночный байт) в текущий поток и перемещает текущую позицию на количество записанных байтов |

Для чтения и записи в память предусмотрен специальный базовый класс .NET по имени *System.IO.MemoryStream*, а для работы с сетевыми данными — класс *System.Net.Sockets.NetworkStream*. Для чтения и записи данных в именованные каналы никаких базовых потоковых классов в .NET не предлагается, но зато имеется один обобщенный потоковый класс System.IO.Stream, от которого можно наследовать нужный класс. Никаких предположений касательно природы внешнего источника данных в классе Stream не делается.

В роли внешнего источника может даже выступать переменная внутри кода. Как бы парадоксально это не звучало, но прием с использованием потоков для передачи данных между переменными может оказываться очень удобным способом для преобразования данных из одного типа в другой. Для преобразования данных из числовых в строковые или для форматирования строк в языке С применялось нечто подобное — функция sprintf.

Преимущество использования отдельного объекта для передачи данных вместо применения для этого классов FileInfo или DirectoryInfo состоит в том, что разделение понятий передачи данных и конкретного источника существенно упрощает изменение этих источников. В самих объектах потоков содержится масса обобщенного кода, касающегося перемещения данных между внешними источниками и переменными в создаваемом самостоятельно коде. Не смешивая этот код с концепцией определенного источника данных, можно значительно повысить возможность его многократного использования в разных обстоятельствах (через наследование).

Например, классы **StringReader** и **StringWriter**являются частью одного и того же дерева наследования, как и те два класса, которые будут применяться чуть позже для чтения и записи текстовых файлов. Все классы почти всегда "за кулисами" будут разделять приличную часть кода.

**Анализ байтовых классов потоков.**

**Анализ символьных классов потоков.**

**Анализ двоичных классов потоков.**

**Использование класса FileStream для файловых операций.**

Класс FileStream предоставляет реализацию абстрактного члена Stream в манере, подходящей для потоковой работы с файлами. Это элементарный поток, и он может записывать или читать только один байт или массив байтов. Однако взаимодействовать с членами типа FileStream придется нечасто. Вместо этого, скорее всего, будут использоваться оболочки потоков, которые облегчают работу с текстовыми данными или типами .NET. Тем не менее, в целях иллюстрации полезно поэкспериментировать с возможностями синхронного чтения/записи типа FileStream.

Экземпляр FileStream применяется для чтения и записи данных в любой файл. Для создания экземпляра FileStream потребуется указать следующие фрагменты информации:

* *Файл*, к которому должен получаться доступ.
* *Режим открытия файла*. Например, планируется создать новый файл или же открыть существующий? Если планируется открыть существующий файл, то должно ли перезаписываться имеющееся в нем содержимое, или новые данные должны добавляться к концу файла?
* *Вид доступа к файлу*. Например, нужен ли доступ для выполнения чтения или записи либо того и другого вместе?
* *Общий доступ*, который показывает, должен ли доступ к файлу быть эксклюзивным или же должна быть возможность доступа со стороны других потоков одновременно. Если да, то разрешено ли другим потокам чтение, запись либо то и другое.

Первый из этих фрагментов информации обычно представлен в виде строки, содержащей полный путь к нужному файлу. Однако помимо строки существуют и дополнительные конструкторы, которые вместо строки принимают файловый дескриптор в стиле Windows API. Остальные три фрагмента информации представляются с помощью трех .NET-перечислений FileMode, FileAccess и FileShare. Значения этих перечислений описаны в таблице:

|  |  |
| --- | --- |
| Перечисление | Значение |
| **FileMode** | Append, Create, CreateNew, Open, OpenOrCreate, Truncate |
| **FileAccess** | Read, ReadWrite, Write |
| **FileShare** | Delete, Inheritable, None, Read, ReadWrite, Write |

Обратите внимание, что в случае FileMode, если запрашивается режим, не соответствующий существующему состоянию файла, может быть сгенерировано исключение. Значения Append, Open и Truncate будут приводить к генерации исключения, если файл не существует, а значение CreateNew — наоборот, если он уже существует. Значения Create и OpenOrCreate подходят в обоих сценариях, но Create приводит к удалению любого существующего файла и замене его новым, изначально пустым.

Перечисления FileAccess и FileShare являются битовыми флагами, поэтому их значения могут комбинироваться с помощью битовой операции "ИЛИ", т.е. |. Для создания FileStream доступно большое количество конструкторов. Три наиболее простейших из них работают так, как описано ниже:

// Создает файл с доступом для чтения и записи и позволяет

// другим потокам получать к нему доступ для чтения

FileStream fs = new FileStream(@"С:\C# Projects\Project.doc", FileMode.Create);

// Делает то же самое, что и предыдущий, но позволяет

// другим потокам получать доступ к файлу для записи

FileStream fs2 = new FileStream(@"С:\C# Projects\Project2.doc",

FileMode.Create, FileAccess.Write);

// Делает то же самое, что и предыдущие конструкторы, но не

// позволяет другим потокам получать доступ к файлу до тех пор,

// пока fs3 остается открытым

FileStream fs3 = new FileStream(@"С:\C# Projects\Project3.doc",

FileMode.Create, FileAccess.Write, FileShare.None);

Как видно в коде, перегруженные версии данных конструкторов способны подставлять стандартные значения FileAccess.ReadWrite и FileShare.Read на месте третьего и четвертого параметров в зависимости от значения FileMode. Можно также создавать файловый поток из экземпляра FileInfo.

После окончания работы поток нужно закрыть:

fs.Close();

Закрытие потока приводит к освобождению всех ассоциированных с ним ресурсов и позволяет другим приложениям запускать потоки для работы с тем же файлом. Кроме того, это действие приводит к очистке буфера. Между открытием и закрытием потока нужно производить собственно чтение и/или запись данных. В классе FileStream для этого предусмотрен набор методов.

Метод **ReadByte()** представляет собой самый простой способ для чтения данных. Он берет один байт из потока и приводит результат к типу int со значением в диапазоне от О до 255. В случае достижения конца потока он возвращает -1.

Если необходимо, чтобы за один раз читалось сразу множество байтов, можно вызывать метод **Read()**, который читает указанное количество байтов в массив. Метод Read() возвращает действительное количество прочитанных байтов; если возвращается значение О, значит, был достигнут конец потока. Ниже показан пример чтения данных в массив байтов по имени ByteArray:

int nBytesRead = fs.Read(ByteArray, 0, nBytes);

Во втором параметре метод Read() принимает значение смещения, которое позволяет указать, что массив должен заполняться, начиная не с первого, а с какого-то другого элемента. В третьем параметре можно указать, сколько байтов должно читаться в массив.

Для выполнения записи данных доступно два метода — **WriteByte()** и **Write()**. Метод WriteByte() позволяет записывать по одному байту в поток:

byte NextByte = 100;

fs.WriteByte(NextByte);

Помимо этих методов в FileStream есть и множество других методов и свойств, имеющих отношение к выполнению вспомогательных операций, таких как определение количества байтов в потоке, блокирование потока и очистка буфера. Для выполнения базовых операций чтения и записи эти дополнительные методы обычно применять не требуется; всю информацию о них можно найти в документации SDK.

В данной статье применение класса FileStream иллюстрируются на примере приложения BinaryFileReader, которое может считывать и отображать любой файл. Создайте в Visual Studio 2010 новый проект типа приложения WPF. Это приложение должно иметь один элемент меню, открывающий стандартное диалоговое окно OpenFileDialog для указания файла, и затем отображать содержимое этого файла в двоичном формате.

Поскольку считываться будут двоичные файлы, необходимо чтобы приложение было способно отображать непечатаемые символы. Для этого каждый байт файла будет отображаться отдельно, с выводом по 16 байтов в каждой строке многострочного текстового поля. В случае если байт представляет собой печатаемый ASCII-символ, будет отображаться этот символ, а если нет, то содержащее в байте значение будет отображаться в шестнадцатеричном формате. И в том и в другом случае отображаемый текст должен дополняться пробелами так, чтобы отображаемый байт занимал четыре столбца.

Используем следующую разметку и код:

<Window x:Class="BinaryFileReader.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="MainWindow" Height="350" Width="525">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="auto"></RowDefinition>

<RowDefinition></RowDefinition>

</Grid.RowDefinitions>

<Menu>

<MenuItem Header="File" Click="MenuItem\_Click"></MenuItem>

</Menu>

<TextBox Margin="0,5,0,5" x:Name="textFile" TextWrapping="Wrap" Grid.Row="1"

VerticalScrollBarVisibility="Auto"></TextBox>

</Grid>

</Window>

using System;

using System.IO;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using Microsoft.Win32;

namespace BinaryFileReader

{

/// <summary>

/// Логика взаимодействия для MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

// Диалоговое окно для отображения файлов

private readonly OpenFileDialog myOpenFileDialog = new OpenFileDialog();

// Путь к файлу

private string pathFile;

public MainWindow()

{

InitializeComponent();

myOpenFileDialog.FileOk += OnOpenFileDialogOK;

}

private void MenuItem\_Click(object sender, RoutedEventArgs e)

{

myOpenFileDialog.ShowDialog();

}

private void OnOpenFileDialogOK(object sender, EventArgs e)

{

pathFile = myOpenFileDialog.FileName;

DisplayFile();

}

void DisplayFile()

{

int nCols = 16;

FileStream fs = new FileStream(pathFile, FileMode.Open, FileAccess.Read);

long nBytesRead = fs.Length;

if (nBytesRead > 65536 / 4)

nBytesRead = 65536 / 4;

int nLines = (int)(nBytesRead / nCols) + 1;

string[] lines = new string[nLines];

int nBytesToRead = 0;

for (int i = 0; i < nLines; i++)

{

StringBuilder nextLine = new StringBuilder();

nextLine.Capacity = 4 \* nCols;

for (int j = 0; j < nLines; j++)

{

int nextByte = fs.ReadByte();

nBytesToRead++;

if (nextByte < 0 || nBytesToRead > 65536)

break;

char nextChar = (char)nextByte;

nextLine.Append(" x0"+string.Format("{0,1:X}", (int)nextChar));

}

lines[i] = nextLine.ToString();

}

fs.Close();

string text = "";

foreach (string l in lines)

text += l;

textFile.Text = text;

}

}

}



**Использование класса StreamWriter и StreamReader для файловых операций.**

Классы StreamWriter и StreamReader удобны во всех случаях, когда нужно читать или записывать символьные данные (например, строки). Оба типа работают по умолчанию с символами Unicode; однако это можно изменить предоставлением правильно сконфигурированной ссылки на объект *System.Text.Encoding*. Чтобы не усложнять пример, предположим, что кодировка по умолчанию Unicode вполне устраивает.

Класс StreamReader, унаследован от абстрактного класса по имени TextReader. Базовый класс предлагает очень ограниченный набор функциональности каждому из его наследников, в частности — возможность читать и "заглядывать" (peek) в символьный поток.

Класс StreamWriter наследуется от абстрактного базового класса по имени TextWriter. В этом классе определены члены, позволяющие производным типам записывать текстовые данные в заданный символьный поток.

Чтобы приблизить вас к пониманию основных возможностей записи классов StreamWriter и StringWriter, ниже представлены описания основных членов абстрактного базового класса TextWriter:

**Close()**

Этот метод закрывает объект-писатель и освобождает все связанные с ним ресурсы. В процессе автоматически сбрасывается буфер (опять-таки, этот член функционально эквивалентен методу Dispose())

**Flush ()**

Этот метод очищает все буферы текущего объекта-писателя и записывает все буферизованные данные на лежащее в основе устройство, однако, не закрывает его

**NewLine**

Это свойство задает константу перевода строки для унаследованного класса писателя. По умолчанию ограничителем строки в Windows является возврат каретки, за которым следует перевод строки (\r\n)

**Write()**

Этот перегруженный метод записывает данные в текстовый поток без добавления константы новой строки

**WriteLine()**

Этот перегруженный метод записывает данные в текстовый поток с добавлением константы новой строки

Последние два члена класса TextWriter, скорее всего, покажутся знакомыми. Если помните, тип System.Console имеет члены Write() и WriteLine(), которые выталкивают текстовые данные на стандартное устройство вывода. Фактически свойство Console.In хранит TextReader, a Console.Out — TextWriter.

Унаследованный класс StreamWriter предоставляет соответствующую реализацию методов Write(), Close() и Flush(), а также определяет дополнительное свойство *AutoFlush*. Когда это свойство установлено в true, оно заставляет StreamWriter выталкивать данные при каждой операции записи. Имейте в виду, что можно обеспечить более высокую производительность, установив AutoFlush в false, но при этом всегда вызывать Close() по завершении работы с StreamWriter.

Используя классы StreamReader и StreamWriter, не нужно беспокоиться о применяемой в файле кодировке (формате текста). К числу возможных кодировок относятся ASCII (1 байт на каждый символ) и Unicode с его разновидностями UTF7, UTF8 и UTF32. Текстовые файлы в системах Windows 9х всегда используют кодировку ASCII, потому что в них Unicode не поддерживается. Но поскольку в ОС Windows NT, 2000, XP, 2003, Vista, Windows Server 2008 и Windows 7 кодировка Unicode поддерживается, теоретически текстовые файлы могут содержать данные не ASCII, а Unicode, UTF7, UTF8 и UTF32.

По соглашению, если файл имеет формат ASCII, он будет просто содержать текст. Если же файл имеет какой-то из форматов Unicode, об этом будет обязательно сообщаться в первых двух или трех байтах файла, в которых устанавливаются определенные комбинации значений для обозначения используемого в файле формата.

Эти байты называют ***маркерами байтовой кодировки***. При открытии файла в любом стандартном приложении Windows, таком как Notepad или WordPad, об этом беспокоиться не нужно, потому что эти приложения различают кодировку, и будут автоматически читать файл правильным образом. Точно так же обстоит дело и с классом StreamReader, который умеет правильно считывать файл в любом из этих форматов, и классом StreamWriter, умеющим форматировать записываемый файл с использованием любой запрашиваемой кодировки. Если для чтения и записи текстового файла планируется использовать класс FileStream, то об этих вещах необходимо заботиться самостоятельно.

**Использование класса BinaryWriter и BinaryReader для файловых операций.**

Для чтения и записи двоичных значении встроенных в C# типов данных служат классы потоков BinaryReader и BinaryWriter. Используя эти потоки, следует иметь в виду, что данные считываются и записываются во внутреннем двоичном формате, а не в удобочитаемой текстовой форме.

Класс BinaryWriter служит оболочкой, в которую заключается байтовый поток, управляющий выводом двоичных данных. Ниже приведен наиболее часто употребляемый конструктор этого класса:

*BinaryWriter(Stream output)*

где output обозначает поток, в который выводятся записываемые данные. Для записи в выходной файл в качестве параметра output может быть указан объект, создаваемый средствами класса FileStream. Если же параметр output оказывается пустым, то генерируется исключение ArgumentNullException. А если поток, определяемый параметром output, не был открыт для записи данных, то генерируется исключение ArgumentException. По завершении вывода в поток типа BinaryWriter его нужно закрыть. При этом закрывается и базовый поток.

Класс BinaryReader служит оболочкой, в которую заключается байтовый поток, управляющий вводом двоичных данных. Ниже приведен наиболее часто употребляемый конструктор этого класса:

*BinaryReader(Stream input)*

где input обозначает поток, из которого вводятся считываемые данные. Для чтения из входного файла в качестве параметра input может быть указан объект, создаваемый средствами класса FileStream.

Если же поток, определяемый параметром input, не был открыт для чтения данных или оказался недоступным по иным причинам, то генерируется исключение ArgumentException. По завершении ввода из потока типа BinaryReader его нужно закрыть. При этом закрывается и базовый поток.

Давайте рассмотрим пример использования данных классов:

using System;

using System.IO;

namespace ConsoleApplication1

{

class Program

{

static void Main()

{

BinaryWriter dataOut;

int i = 10;

double d = 1023.56;

bool b = true;

string str = "Это текст";

// Открыть файл для ввода

try

{

dataOut = new BinaryWriter(new FileStream("C:/testData",FileMode.Create));

}

catch (IOException ex)

{

Console.WriteLine("Ошибка открытия файла: " + ex.Message);

return;

}

// Записать данные в файл

try

{

Console.WriteLine("Запись " + i);

dataOut.Write(i);

Console.WriteLine("Запись " + d);

dataOut.Write(d);

Console.WriteLine("Запись "+b);

dataOut.Write(b);

Console.WriteLine("Запись "+str);

dataOut.Write(str);

}

catch (IOException ex)

{

Console.WriteLine("Ошибка ввода-вывода файла: " + ex.Message);

return;

}

finally

{

dataOut.Close();

}

}

}

}

**Использование классов Directory, DirectoryInfo, FileInfo для файловых операций.**

Directory - Предоставляет статические методы для создания, перемещения и перечисления в каталогах и вложенных каталогах. Этот класс не наследуется.

public static class Directory

В следующем примере показано, как извлечь все текстовые файлы из каталога и переместить их в новый каталог. После перемещения файлы больше не существуют в исходном каталоге.

using System;

using System.IO;

namespace ConsoleApplication

{

class Program

{

static void Main(string[] args)

{

string sourceDirectory = @"C:\current";

string archiveDirectory = @"C:\archive";

try

{

var txtFiles = Directory.EnumerateFiles(sourceDirectory, "\*.txt");

foreach (string currentFile in txtFiles)

{

string fileName = currentFile.Substring(sourceDirectory.Length + 1);

Directory.Move(currentFile, Path.Combine(archiveDirectory, fileName));

}

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

}

В следующем примере показано, как использовать метод [EnumerateFiles](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory.enumeratefiles?view=netcore-3.1) для получения коллекции текстовых файлов из каталога, а затем использовать эту коллекцию в запросе для поиска всех строк, содержащих "example

using System;

using System.IO;

using System.Linq;

namespace ConsoleApplication

{

class Program

{

static void Main(string[] args)

{

string archiveDirectory = @"C:\archive";

var files = from retrievedFile in Directory.EnumerateFiles(archiveDirectory, "\*.txt", SearchOption.AllDirectories)

from line in File.ReadLines(retrievedFile)

where line.Contains("Example")

select new

{

File = retrievedFile,

Line = line

};

foreach (var f in files)

{

Console.WriteLine("{0} contains {1}", f.File, f.Line);

}

Console.WriteLine("{0} lines found.", files.Count().ToString());

}

}

}

В следующем примере показано, как переместить каталог и все его файлы в новый каталог. Исходный каталог больше не существует после перемещения.

using System;

using System.IO;

namespace ConsoleApplication

{

class Program

{

static void Main(string[] args)

{

string sourceDirectory = @"C:\source";

string destinationDirectory = @"C:\destination";

try

{

Directory.Move(sourceDirectory, destinationDirectory);

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

}

Следует использовать класс [Directory](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory?view=netcore-3.1) для стандартных операций, таких как копирование, перемещение, переименование, создание и удаление каталогов.

* Чтобы создать каталог, используйте один из методов [CreateDirectory](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory.createdirectory?view=netcore-3.1).
* Чтобы удалить каталог, используйте один из методов [Delete](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory.delete?view=netcore-3.1).
* Чтобы получить или задать текущий каталог для приложения, используйте метод [GetCurrentDirectory](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory.getcurrentdirectory?view=netcore-3.1) или [SetCurrentDirectory](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory.setcurrentdirectory?view=netcore-3.1).
* Чтобы управлять [DateTime](https://docs.microsoft.com/ru-ru/dotnet/api/system.datetime?view=netcore-3.1) сведениями, связанными с созданием, доступом и записью каталога, используйте такие методы, как [SetLastAccessTime](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory.setlastaccesstime?view=netcore-3.1) и [SetCreationTime](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory.setcreationtime?view=netcore-3.1).

Статические методы класса [Directory](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory?view=netcore-3.1) выполняют проверку безопасности для всех методов. Если вы собираетесь повторно использовать объект несколько раз, рассмотрите возможность использования соответствующего метода экземпляра [DirectoryInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directoryinfo?view=netcore-3.1) вместо этого, так как проверка безопасности не всегда будет требоваться.

Если вы выполняете только одно действие, связанное с каталогом, возможно, более эффективно использовать статический [Directory](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory?view=netcore-3.1) метод, а не соответствующий метод экземпляра [DirectoryInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directoryinfo?view=netcore-3.1). Для большинства методов [Directory](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory?view=netcore-3.1) требуется путь к каталогу, к которому выполняется управление.

**DirectoryInfo -** Предоставляет методы экземпляра класса для создания, перемещения и перечисления в каталогах и подкаталогах. Этот класс не наследуется.

public sealed class DirectoryInfo : System.IO.FileSystemInfo

В следующем примере показаны некоторые из основных элементов класса DirectoryInfo.

using System;

using System.IO;

class Test

{

public static void Main()

{

// Specify the directories you want to manipulate.

DirectoryInfo di = new DirectoryInfo(@"c:\MyDir");

try

{

// Determine whether the directory exists.

if (di.Exists)

{

// Indicate that the directory already exists.

Console.WriteLine("That path exists already.");

return;

}

// Try to create the directory.

di.Create();

Console.WriteLine("The directory was created successfully.");

// Delete the directory.

di.Delete();

Console.WriteLine("The directory was deleted successfully.");

}

catch (Exception e)

{

Console.WriteLine("The process failed: {0}", e.ToString());

}

finally {}

}

}

В следующем примере показано, как скопировать каталог и его содержимое.

using System;

using System.IO;

class CopyDir

{

public static void CopyAll(DirectoryInfo source, DirectoryInfo target)

{

if (source.FullName.ToLower() == target.FullName.ToLower())

{

return;

}

// Check if the target directory exists, if not, create it.

if (Directory.Exists(target.FullName) == false)

{

Directory.CreateDirectory(target.FullName);

}

// Copy each file into it's new directory.

foreach (FileInfo fi in source.GetFiles())

{

Console.WriteLine(@"Copying {0}\{1}", target.FullName, fi.Name);

fi.CopyTo(Path.Combine(target.ToString(), fi.Name), true);

}

// Copy each subdirectory using recursion.

foreach (DirectoryInfo diSourceSubDir in source.GetDirectories())

{

DirectoryInfo nextTargetSubDir =

target.CreateSubdirectory(diSourceSubDir.Name);

CopyAll(diSourceSubDir, nextTargetSubDir);

}

}

public static void Main()

{

string sourceDirectory = @"c:\sourceDirectory";

string targetDirectory = @"c:\targetDirectory";

DirectoryInfo diSource = new DirectoryInfo(sourceDirectory);

DirectoryInfo diTarget = new DirectoryInfo(targetDirectory);

CopyAll(diSource, diTarget);

}

// Output will vary based on the contents of the source directory.

}

Следует использовать класс [DirectoryInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directoryinfo?view=netcore-3.1) для стандартных операций, таких как копирование, перемещение, переименование, создание и удаление каталогов.

Если вы собираетесь повторно использовать объект несколько раз, рассмотрите возможность использования метода экземпляра [DirectoryInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directoryinfo?view=netcore-3.1) вместо соответствующих статических методов класса [Directory](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.directory?view=netcore-3.1), поскольку проверка безопасности не всегда требуется.

В членах, принимающих путь в качестве входной строки, этот путь должен иметь правильный формат или возникнет исключение. Например, если путь является полным, но начинается с пробела, путь не усекается в методах класса. Поэтому путь имеет неправильный формат и возникает исключение. Аналогично, путь или сочетание путей не могут быть полными и дважды. Например, "c:\temp c:\Windows" также вызывает исключение в большинстве случаев. При использовании методов, принимающих строку пути, убедитесь, что пути имеют правильный формат.

В членах, принимающих путь, путь может ссылаться на файл или только на каталог. Указанный путь может также ссылаться на относительный путь или путь в формате UNC для имени сервера и общего ресурса. Например, все следующие допустимые пути:

* "c:\\Мидир\\Мифиле.ткст" in C#или "к:\мидир\мифиле.ткст" в Visual Basic.
* "c:\\Мидир" in C#или "к:\мидир" в Visual Basic.
* "MyDir\\Мисубдир" in C#или "мидир\мисубдир" в Visual Basic.
* "\\\\Мисервер\\Мишаре" in C#или "\\MyServer\MyShare" в Visual Basic.

По умолчанию всем пользователям предоставляется полный доступ на чтение и запись к новым каталогам.

**FileInfo -** Предоставляет свойства и методы экземпляра для создания, копирования, удаления, перемещения и открытия файлов, а также позволяет создавать объекты [FileStream](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.filestream?view=netcore-3.1). Этот класс не наследуется.

public sealed class FileInfo : System.IO.FileSystemInfo

В следующем примере показаны некоторые из основных элементов класса FileInfo.

При первом извлечении свойств [FileInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.fileinfo?view=netcore-3.1) вызывает метод [Refresh](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.filesysteminfo.refresh?view=netcore-3.1) и кэширует сведения о файле. При последующих вызовах необходимо вызвать [Refresh](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.filesysteminfo.refresh?view=netcore-3.1), чтобы получить последнюю копию информации.

using System;

using System.IO;

class Test

{

public static void Main()

{

string path = Path.GetTempFileName();

var fi1 = new FileInfo(path);

// Create a file to write to.

using (StreamWriter sw = fi1.CreateText())

{

sw.WriteLine("Hello");

sw.WriteLine("And");

sw.WriteLine("Welcome");

}

// Open the file to read from.

using (StreamReader sr = fi1.OpenText())

{

var s = "";

while ((s = sr.ReadLine()) != null)

{

Console.WriteLine(s);

}

}

try

{

string path2 = Path.GetTempFileName();

var fi2 = new FileInfo(path2);

// Ensure that the target does not exist.

fi2.Delete();

// Copy the file.

fi1.CopyTo(path2);

Console.WriteLine($"{path} was copied to {path2}.");

// Delete the newly created file.

fi2.Delete();

Console.WriteLine($"{path2} was successfully deleted.");

}

catch (Exception e)

{

Console.WriteLine($"The process failed: {e.ToString()}");

}

}

}

В этом примере создаются выходные данные, аналогичные приведенным ниже.

Hello

And

Welcome

C:\Users\userName\AppData\Local\Temp\tmp70AB.tmp was copied to C:\Users\userName\AppData\Local\Temp\tmp70CB.tmp.

C:\Users\userName\AppData\Local\Temp\tmp70CB.tmp was successfully deleted.

Используйте класс [FileInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.fileinfo?view=netcore-3.1) для выполнения типичных операций, таких как копирование, перемещение, переименование, создание, открытие, удаление и добавление в файлы.

Если вы выполняете несколько операций над одним и тем же файлом, более эффективно использовать [FileInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.fileinfo?view=netcore-3.1) методы экземпляра вместо соответствующих статических методов класса [File](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.file?view=netcore-3.1), поскольку проверка безопасности не всегда будет требоваться.

Многие из [FileInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.fileinfo?view=netcore-3.1) методов возвращают другие типы ввода-вывода при создании или открытии файлов. Эти другие типы можно использовать для дальнейшего управления файлом.

По умолчанию всем пользователям предоставляется полный доступ на чтение и запись к новым файлам.

В следующей таблице описаны перечисления, используемые для настройки поведения различных методов [FileInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.fileinfo?view=netcore-3.1).

| **Перечисление** | **Описание** |
| --- | --- |
| [FileAccess](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.fileaccess?view=netcore-3.1) | Указывает доступ к файлу для чтения и записи. |
| [FileShare](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.fileshare?view=netcore-3.1) | Указывает уровень доступа, разрешенный для уже используемого файла. |
| [FileMode](https://docs.microsoft.com/ru-ru/dotnet/api/system.io.filemode?view=netcore-3.1) | Указывает, будет ли содержимое существующего файла сохранено или перезаписано, и должны ли запросы на создание существующего файла вызывать исключение. |

**Регулярные выражения.**

**Регулярные выражения** — это часть небольшой технологической области, невероятно широко используемой в огромном диапазоне программ. Регулярные выражения можно представить себе как мини-язык программирования, имеющий одно специфическое назначение: находить подстроки в больших строковых выражениях.

Это не новая технология, изначально она появилась в среде **UNIX** и обычно используется в языке программирования Perl. Разработчики из Microsoft перенесли ее в Windows, где до недавнего времени эта технология применялась в основном со сценарными языками. Однако теперь регулярные выражения поддерживаются множеством классов .NET из пространства имен System.Text.RegularExpressions. Случаи применения регулярных выражений можно встретить во многих частях среды .NET Framework. В частности, вы найдете их в серверных элементах управления проверкой ASP.NET.

## Введение в регулярные выражения

Язык регулярных выражений предназначен специально для обработки строк. Он включает два средства:

1. Набор управляющих кодов для идентификации специфических типов символов
2. Система для группирования частей подстрок и промежуточных результатов таких действий

С помощью регулярных выражений можно выполнять достаточно сложные и высокоуровневые действия над строками:

* Идентифицировать (и возможно, помечать к удалению) все повторяющиеся слова в строке
* Сделать заглавными первые буквы всех слов
* Преобразовать первые буквы всех слов длиннее трех символов в заглавные
* Обеспечить правильную капитализацию предложений
* Выделить различные элементы в URI (например, имея http://www.professorweb.ru, выделить протокол, имя компьютера, имя файла и т.д.)

Главным преимуществом регулярных выражений является использование **метасимволов** — специальные символы, задающие команды, а также управляющие последовательности, которые работают подобно управляющим последовательностям C#. Это символы, предваренные знаком обратного слеша (\) и имеющие специальное назначение.

В следующей таблице специальные метасимволы регулярных выражений C# сгруппированы по смыслу:

|  |  |  |  |
| --- | --- | --- | --- |
| Метасимволы, используемые в регулярных выражениях C# | | | |
| **Символ** | **Значение** | **Пример** | **Соответствует** |
| Классы символов | | | |
| [...] | Любой из символов, указанных в скобках | [a-z] | В исходной строке может быть любой символ английского алфавита в нижнем регистре |
| [^...] | Любой из символов, не указанных в скобках | [^0-9] | В исходной строке может быть любой символ кроме цифр |
| . | Любой символ, кроме перевода строки или другого разделителя Unicode-строки |  |  |
| \w | Любой текстовый символ, не являющийся пробелом, символом табуляции и т.п. |  |  |
| \W | Любой символ, не являющийся текстовым символом |  |  |
| \s | Любой пробельный символ из набора Unicode |  |  |
| \S | Любой непробельный символ из набора Unicode. Обратите внимание, что символы \w и \S - это не одно и то же |  |  |
| \d | Любые ASCII-цифры. Эквивалентно [0-9] |  |  |
| \D | Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9] |  |  |
| **Символы повторения** | | | |
| {n,m} | Соответствует предшествующему шаблону, повторенному не менее n и не более m раз | s{2,4} | "Press", "ssl", "progressss" |
| {n,} | Соответствует предшествующему шаблону, повторенному n или более раз | s{1,} | "ssl" |
| {n} | Соответствует в точности n экземплярам предшествующего шаблона | s{2} | "Press", "ssl", но не "progressss" |
| ? | Соответствует нулю или одному экземпляру предшествующего шаблона; предшествующий шаблон является необязательным | Эквивалентно {0,1} |  |
| + | Соответствует одному или более экземплярам предшествующего шаблона | Эквивалентно {1,} |  |
| \* | Соответствует нулю или более экземплярам предшествующего шаблона | Эквивалентно {0,} |  |
| **Символы регулярных выражений выбора** | | | |
| | | Соответствует либо подвыражению слева, либо подвыражению справа (аналог логической операции ИЛИ). |  |  |
| (...) | Группировка. Группирует элементы в единое целое, которое может использоваться с символами \*, +, ?, | и т.п. Также запоминает символы, соответствующие этой группе для использования в последующих ссылках. |  |  |
| (?:...) | Только группировка. Группирует элементы в единое целое, но не запоминает символы, соответствующие этой группе. |  |  |
| **Якорные символы регулярных выражений** | | | |
| ^ | Соответствует началу строкового выражения или началу строки при многострочном поиске. | ^Hello | "Hello, world", но не "Ok, Hello world" т.к. в этой строке слово "Hello" находится не в начале |
| $ | Соответствует концу строкового выражения или концу строки при многострочном поиске. | Hello$ | "World, Hello" |
| \b | Соответствует границе слова, т.е. соответствует позиции между символом \w и символом \W или между символом \w и началом или концом строки. | \b(my)\b | В строке "Hello my world" выберет слово "my" |
| \B | Соответствует позиции, не являющейся границей слов. | \B(ld)\b | Соответствие найдется в слове "World", но не в слове "ld" |

## Использование регулярных выражений в C#

Безуcловно, задачу поиска и замены подстроки в строке можно решить на C# с использованием различных методов [System.String](https://professorweb.ru/my/csharp/charp_theory/level4/4_7.php) и [System.Text.StringBuilder](https://professorweb.ru/my/csharp/charp_theory/level4/4_8.php). Однако в некоторых случаях это потребует написания большого объема кода C#. Если вы используете регулярные выражения, то весь этот код сокращается буквально до нескольких строк. По сути, вы создаете экземпляр объекта **RegEx**, передаете ему строку для обработки, а также само регулярное выражение (строку, включающую инструкции на языке регулярных выражений) — и все готово.

В следующей таблице показана часть информации о перечислении RegexOptions, экземпляр которого можно передать конструктору класса RegEx:

|  |  |
| --- | --- |
| Структура перечисления RegexOptions | |
| **Член** | **Описание** |
| CultureInvariant | Предписывает игнорировать национальные установки строки |
| ExplicitCapture | Модифицирует способ поиска соответствия, обеспечивая только буквальное соответствие |
| IgnoreCase | Игнорирует регистр символов во входной строке |
| IgnorePatternWhitespace | Удаляет из строки не защищенные управляющими символами пробелы и разрешает комментарии, начинающиеся со знака фунта или хеша |
| Multiline | Изменяет значение символов ^ и $ так, что они применяются к началу и концу каждой строки, а не только к началу и концу всего входного текста |
| RightToLeft | Предписывает читать входную строку справа налево вместо направления по умолчанию — слева направо (что удобно для некоторых азиатских и других языков, которые читаются в таком направлении) |
| Singleline | Специфицирует однострочный режим, в котором точка (.) символизирует соответствие любому символу |

После создания шаблона регулярного выражения с ним можно осуществить различные действия, в зависимости от того, что вам необходимо. Можно просто проверить, существует ли текст, соответствующий шаблону, в исходной строке. Для этого нужно использовать метод **IsMatch()**, который возвращает логическое значение:

using System;

using System.Text.RegularExpressions;

class Example

{

static void Main()

{

// Массив тестируемых строк

string[] test = {

"Wuck World", "Hello world", "My wonderful world"

};

// Проверим, содержится ли в исходных строках слово World

// при этом мы не укажем опции RegexOption

Regex regex = new Regex("World");

Console.WriteLine("Регистрозависимый поиск: ");

foreach (string str in test)

{

if (regex.IsMatch(str))

Console.WriteLine("В исходной строке: \"{0}\" есть совпадения!", str);

}

Console.WriteLine();

// Теперь укажем поиск, не зависимый от регистра

regex = new Regex("World", RegexOptions.IgnoreCase);

Console.WriteLine("РегистроНЕзависимый поиск: ");

foreach (string str in test)

{

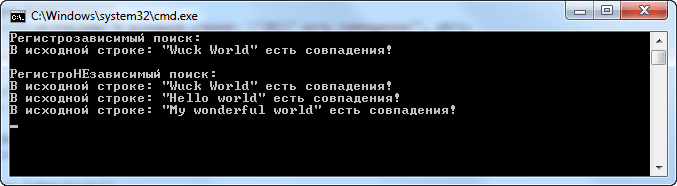
if (regex.IsMatch(str))

Console.WriteLine("В исходной строке: \"{0}\" есть совпадения!", str);

}

}

}



Если нужно вернуть найденное соответствие из исходной строки, то можно воспользоваться методом **Match()**, который возвращает объект класса Match, содержащий сведения о первой подстроке, которая сопоставлена шаблону регулярного выражения. В этом классе имеется свойство Success, которое возвращает значение true, если найдено следующее совпадение, которое можно получить с помощью вызова метода Match.NextMatch(). Эти вызовы метода можно продолжать пока свойство Match.Success не вернет значение false. Например:

using System;

using System.Text.RegularExpressions;

class Example

{

static void Main()

{

// Допустим в исходной строке нужно найти все числа,

// соответствующие стоимости продукта

string input = "Добро пожаловать в наш магазин, вот наши цены: " +

"1 кг. яблок - 20 руб. " +

"2 кг. апельсинов - 30 руб. " +

"0.5 кг. орехов - 50 руб.";

string pattern = @"\b(\d+\W?руб)";

Regex regex = new Regex(pattern);

// Получаем совпадения в экземпляре класса Match

Match match = regex.Match(input);

// отображаем все совпадения

while (match.Success)

{

// Т.к. мы выделили в шаблоне одну группу (одни круглые скобки),

// ссылаемся на найденное значение через свойство Groups класса Match

Console.WriteLine(match.Groups[1].Value);

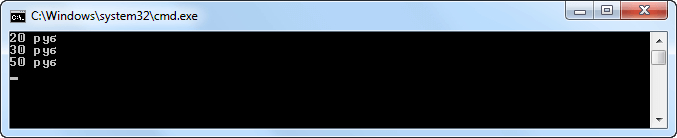
// Переходим к следующему совпадению

match = match.NextMatch();

}

}

}



Извлечь все совпадения можно и более простым способом, используя метод Regex.Matches(), который возвращает объект класса **MatchCollection**, который, в свою очередь, содержит сведения обо всех совпадениях, которые обработчик регулярных выражений находит во входной строке. Например, предыдущий пример может быть переписан для вызова метода Matches вместо метода Match и метода NextMatch:

using System;

using System.Text.RegularExpressions;

class Example

{

static void Main()

{

// Допустим в исходной строке нужно найти все числа,

// соответствующие стоимости продукта

string input = "Добро пожаловать в наш магазин, вот наши цены: " +

"1 кг. яблок - 20 руб. " +

"2 кг. апельсинов - 30 руб. " +

"0.5 кг. орехов - 50 руб.";

string pattern = @"\b(\d+\W?руб)";

Regex regex = new Regex(pattern);

// Достигаем того же результата что и в предыдущем примере,

// используя метод Regex.Matches() возвращающий MatchCollection

foreach (Match match in regex.Matches(input))

{

Console.WriteLine(match.Groups[1].Value);

}

}

}

Наконец, можно не просто извлекать совпадения в исходной строке, но и заменять их на собственные значения. Для этого используется метод Regex.Replace(). В качестве замены методу Replace() можно передавать как строку, так и шаблон замены. В следующей таблице показано как формируются метасимволы для замены:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метасимволы замены в регулярных выражениях C# | | | | |
| **Символ** | **Описание** | **Пример шаблона** | **Пример шаблона замены** | **Результат (входная -> результирующая строки)** |
| $ number | Замещает часть строки, соответствующую группе number | \b(\w+)(\s)(\w+)\b | $3$2$1 | "один два" -> "два один" |
| $$ | Подставляет литерал "$" | \b(\d+)\s?USD | $$$1 | "103 USD" -> "$103" |
| $& | Замещает копией полного соответствия | (\$\*(\d\*(\.+\d+)?){1}) | \*\*$& | "$1.30" -> "\*\*$1.30\*\*" |
| $` | Замещает весь текст входной строки до соответствия | B+ | $` | "AABBCC" -> "AAAACC" |
| $' | Замещает весь текст входной строки после соответствия | B+ | $' | "AABBCC" -> "AACCCC" |
| $+ | Замещает последнюю захваченную группу | B+(C+) | $+ | "AABBCCDD" -> "AACCDD" |
| $\_ | Замещает всю входную строку | B+ | $\_ | "AABBCC" -> "AAAABBCCCC" |

Давайте рассмотрим метод Regex.Replace() на примере:

using System;

using System.Text.RegularExpressions;

class Example

{

static void Main()

{

// Допустим в исходной строке нужно заменить "руб." на "$",

// а стоимость переместить после знака $

string input = "Добро пожаловать в наш магазин, вот наши цены: \n" +

"\t 1 кг. яблок - 20 руб. \n" +

"\t 2 кг. апельсинов - 30 руб. \n" +

"\t 0.5 кг. орехов - 50 руб. \n";

Console.WriteLine("Исходная строка:\n {0}", input);

// В шаблоне используются 2 группы

string pattern = @"\b(\d+)\W?(руб.)";

// Строка замены "руб." на "$"

string replacement1 = "$$$1"; // Перед первой группой ставится знак $,

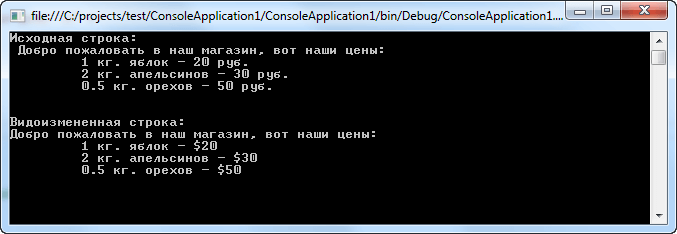
// вторая группа удаляется без замены

input = Regex.Replace(input, pattern, replacement1);

Console.WriteLine("\nВидоизмененная строка: \n" +input);

}

}



рассмотрим еще один пример использования регулярных выражений, где будем искать в исходном тексте слово «сериализация» и его однокоренные слова, при этом выделяя в консоли их другим цветом:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

string myText = @"Сериализация представляет собой процесс сохранения объекта на диске.

В другой части приложения или даже в совершенно отдельном приложении может производиться

десериализация объекта, возвращающая его в состояние, в котором он пребывал до сериализации.";

const string myReg = "со";

MatchCollection myMatch = Regex.Matches(myText,myReg);

Console.WriteLine("Все вхождения строки \"{0}\" в исходной строке: ",myReg);

foreach (Match i in myMatch)

Console.Write("\t"+i.Index);

// Усложним шаблон регулярного выражения

// введя в него специальные метасимволы

const string myReg1 = @"\b[с,д]\S\*ериализац\S\*";

MatchCollection match1 = Regex.Matches(myText,myReg1,RegexOptions.IgnoreCase);

findMyText(myText,match1);

Console.ReadLine();

}

static void findMyText(string text, MatchCollection myMatch)

{

Console.WriteLine("\n\nИсходная строка:\n\n{0}\n\nВидоизмененная строка:\n",text);

// Реализуем выделение ключевых слов в консоли другим цветом

for (int i = 0; i < text.Length; i++)

{

foreach (Match m in myMatch)

{

if ((i >= m.Index) && (i < m.Index+m.Length))

{

Console.BackgroundColor = ConsoleColor.Green;

Console.ForegroundColor = ConsoleColor.Black;

break;

}

else

{

Console.BackgroundColor = ConsoleColor.Black;

Console.ForegroundColor = ConsoleColor.White;

}

}

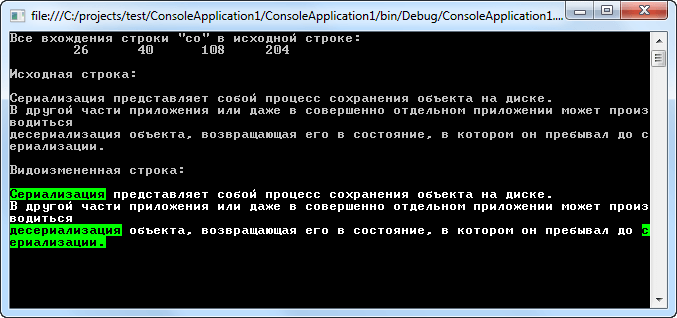
Console.Write(text[i]);

}

}

}

}



Для проверки гибкости работы регулярных выражений, подставьте в исходный текст еще несколько слов «сериализация», вы увидите, что они будут автоматически выделены зеленым цветом в консоли.