Занятие 4, ч2. Перегрузка, обработка исключений, управление ресурсами в .Net

Тренер: Алексей Дышлевой

Основные вопросы

- Перевантаження операцій. Типи й формати перевантажених операцій. Операнди та їх порядок. Операції, що допускають перевантаження. Перевантаження арифметичних операцій, операцій порівняння, операцій перетворення, булевих операцій. операцій присвоювання.
- Обробка виключень. Генерація виключень. Необроблені виключення. Оператори try, catch, finally. Повторна генерація виключень. Виключення у блоці finally. Виключення у фіналізаторах. Виключення у конструкторах. Обмежені області виконання. Критичні фіналі затори та SafeHandle. Користувацькі класи виключень. Забезпечення відкочування змін.
- Життєвий цикл об’єкта. Керовані ресурси в .Net. Клас GC. Деструктор.
- Управління ресурсами. Знищення об’єктів. Фіналізатори. Детерміноване знищення. Інтерфейс IDisposable
- Тип System.Object. Интерфейс IClonable.
- Еквівалентність типів. Інтерфейс IComparable

Интерфейс IComparable

- Используется для того, чтобы упорядочить или отсортировать экземпляры типа, который реализует данный интерфейс
- CompareTo –сравнивает текущий экземпляр с другим объектом того же типа. Возвращает целое число, которое показывает, расположен ли текущий экземпляр перед, после или на той же позиции в порядке сортировки, что и другой объект
  - < 0 – данный экземпляр следует перед сравниваемым объектом
  - 0 – данный экземпляр имеет ту же позицию, что и сравниваемый объект
  - > 0 – данный экземпляр следует после сравниваемым объектом

Пример IComparable

```
public class Temperature : IComparable {
public double temperatureC {get; set; };
   public int CompareTo(object obj) {
      if (obj == null) return 1;
         Temperature otherTemperature = obj as Temperature;
         if (otherTemperature != null)
            return this.temperatureC.CompareTo(otherTemperature.temperatureC);
         else
            throw new ArgumentException("Object is not a Temperature");
 }
public class CompareTemperatures {
    public static void Main() {
         ArrayList temperatures = new ArrayList();
         // Initialize random number generator.
         Random rnd = new Random();
         // Generate 10 temperatures between 0 and 100 randomly.
         for (int ctr = 1; ctr <= 10; ctr++) {
            int degrees = rnd.Next(0, 100);
            Temperature temp = new Temperature();
            temp.TemperatureC = degrees;
            temperatures.Add(temp);
         }
        // Sort ArrayList.
        temperatures.Sort();
        foreach (Temperature temp in temperatures)
           Console.WriteLine(temp.temeratureC);
    }
}
```

Интерфейс IComparer

- Роль **IComparer** заключается в создании дополнительных механизмов сравнения. Например, может потребоваться упорядочивание класса по нескольким полям или свойствам, по возрастанию или убыванию по одному полю, или и то, и другое.
- Как правило, реализуется классами, которыми невозможно реализовать IComparable.
- Кроме того, IComparer используют для сравнения двух объектов одного класса в другом, а IComparabe для сравнения текущего объекта с некоторым другим.
- метод **IComparer.Compare** требует третичного сравнения. 1, 0 или -1 возвращается в зависимости от того, является ли одно значение больше, равно или меньше другого.

Задача

- Реализовать интерфейс IComparable для сравнения двух отрезков

Еще немного об Object

- Object.Equals(object obj) – true, если заданный объект равен текущему; в противном случае – false
- Object.GetHashCode(). Хэш-код – числовое значение, которое используется для идентификации объекта во время проверки равенства; может служить индексом для коллекции. Реализацию этого метода по умолчанию не следует использовать для хэширования в качестве уникального идентификатора объекта.
- Object.ToString() - преобразует объект в строковое представление для отображения

Пример Equals()

```
public override bool Equals(object obj)
 {
 // If this and obj do not refer to the same type, then
they are not equal.
 if (obj.GetType() != this.GetType()) return false;
// Return true if x and y fields match.
Point other = (Point) obj;
return (this.x == other.x) && (this.y == other.y);
 }
```

Пример GetHashCode()

```
// Return the XOR of the x and y fields.
 public override int GetHashCode()
 {
    return x ^ y;
 }
```

Пример ToString()

```
public override String ToString()
{
    return String.Format("({0}, {1})", x, y);
 }
```

Интерфейс IClonable

- Поддерживает копирование, при котором создается новый экземпляр класса с тем же значением, что и у существующего экземпляра
- Используется метод Clone(), который создает новый объект, являющийся копией текущего экземпляра

Задача

- Дополнить пример предыдущего занятия (сравнение отрезков) переопределением Equals(), GetHashCode(), ToString() и перегрузкой !=, ==. Также реализовать интерфейс IClonable.

Жизненный цикл объекта

- При создании объекта 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.

Жизненный цикл объекта

- Когда работа с объектом завершена, он может быть уничтожен. Для возвращения любых ресурсов, используемых этим объектом, используется уничтожение (деструкция). Подобно созданию уничтожение это двухэтапный процесс:
  - Объект очищается, например, путем освобождения любых неуправляемых ресурсов, используемых приложением, таких как дескрипторы файлов или подключения к базам данных.
  - Память, используемая объектом возвращается.
- Контролировать можно только первый из этих этапов – очистку объекта и освобождения ресурсов. Данный шаг можно регулировать за счет реализации деструктора.
- CLR управляет освобождением памяти, используемой управляемыми объектами, однако, если используются неуправляемые объекты, может потребоваться вручную высвобождать память, используемую этими элементами.

Управляемые ресурсы в .Net. Значимые типы

- .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.

Управляемые ресурсы в .Net. Значимые типы

- Коротко:

Значимые типы

- Создаются в стеке
- Когда объект в стеке выходит из области видимости, память, используемая этим объектом, будет немедленно освобождена
- Новые элементы перезаписывают старые, память освобождается автоматически, фрагментация памяти невозможна
- Память быстро выделяется и освобождается. Управление памятью является относительно недорогой операцией

Управляемые ресурсы в .Net. Ссылочные типы

- 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 высвободятся только тогда, когда последняя ссылка на объект исчезнет. Следовательно, время жизни объекта не регулируется рамками какой-либо одной ссылки на этот объект.

Управляемые ресурсы в .Net. Ссылочные типы

- Коротко:

Ссылочные типы

- Размещаются в куче
- Ссылочный тип может иметь несколько ссылок на один и тот же объект
- Объект может быть уничтожен, его деструктор сработает, а его ресурсы высвободятся только тогда, когда последняя ссылка на объект исчезнет
- Может приводить к фрагментации памяти
- Управляются сборщиком мусора
- Более дорогие в управлении

Сборщик мусора

- 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), который показывает, где точно будет размещаться следующий объект
- 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.
- 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.
- 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.

Как работает сборщик мусора

- Сборщик мусора освобождает ресурсы и память для объектов, хранящихся в куче
- Сборщик мусора работает в своем собственном потоке и обычно запускается автоматически
- Когда сборщик мусора работает, другие потоки приложения прекращают работать

Возвращение ресурсов

Сборщик мусора

- Отмечает недостижимые объекты
- Отмечает используемые объекты как достижимые (рекурсивное выполнение)
- Добавляет финализируемые недостижимые объекты во freachable очередь. Freachable очередь хранит указатели на объекты, требующие завершения до восстановления их ресурсов
- Удаляет недостижимые объекты и дефрагментирует кучу
- Обновляет указатели. Объекты, отмеченные как достижимые перемещаются вниз кучи для формирования непрерывного блока. Ссылки на объекты, перемещенные сборщиком мусора, обновляются.
- Возобновляет потоки
- Запускает поток финализации для freachable очереди. После завершения объекта, указатель на этот объект удаляется из freachable очереди.

Корневые элементы приложения

- Корневым элементом (root) называется ячейка в памяти, в которой содержится ссылка на размещающийся в куче объект
- 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
- Для определения корневых элементов приложения среда CLR создает графы объектов, представляющие все достижимые для приложения объекты в куче
- Сборщик мусора никогда не будет создавать граф для одного и того же объекта дважды, избегая необходимости выполнения подсчета циклических ссылок.

Пример работы сборщика мусора

Оптимизация процесса удаления

- **Поколение 0** Новый созданный объект, который еще ни разу не помечался, как кандидат на удаление
- **Поколение 1** Объект, который уже пережил сборку мусора 1 раз. Обычно сюда попадают объекты, которые были помечены для удаления, но не были удалены из-за достаточного свободного места в куче
- **Поколение 2** Объекты, которые пережили более одной очистки памяти сборщиком мусора

Пример работы сборщика мусора разных поколений объектов

Класс GC

- Большую часть времени нужно позволить сборщику мусора выполнять операции в свое время по указанию CLR
- 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

Методы класса 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 используется, 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 выполнения дважды кода завершения. GC.SuppressFinalize(this);
ReRegisterForFinalize	Запрашивает финализатор для объекта, который либо уже завершен или завершение было подавлено.	Метод используется, если есть подавленный 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 следует 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);

Ограничения финализаторов

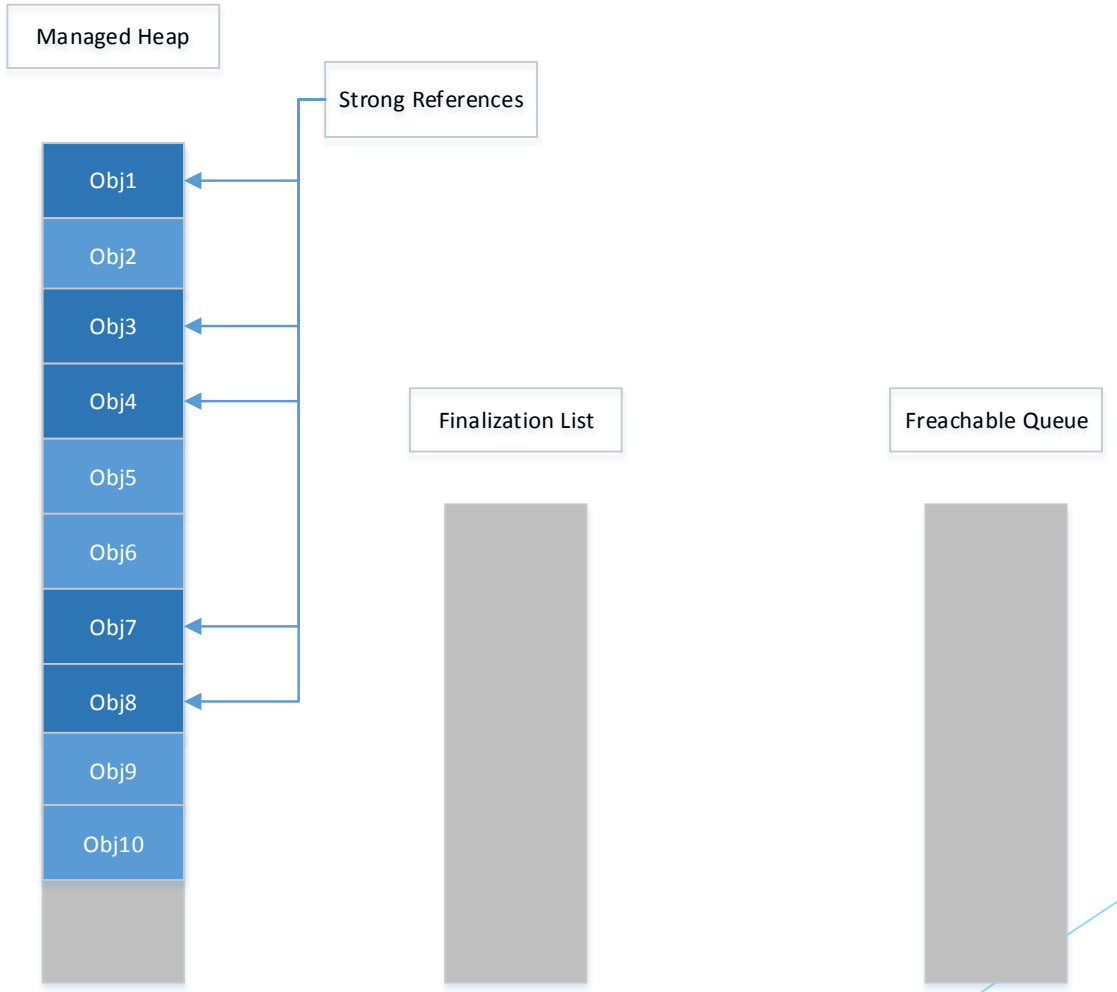
- Нельзя добавить деструктор структуре или любому другому типу значения
- Нельзя объявить модификатор доступа для деструктора (являются неявно защищенными)
- Нельзя объявить деструктор, принимающий параметры
- Класс может иметь только один деструктор
- Деструкторы не могут наследоваться или перегружаться

Механизм использования финализатора

- 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). Этот список финализации представляет собой просматриваемую сборщиком мусора таблицу, где перечислены объекты, которые перед удалением из кучи должны быть обязательно финализированы.
- 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). 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.

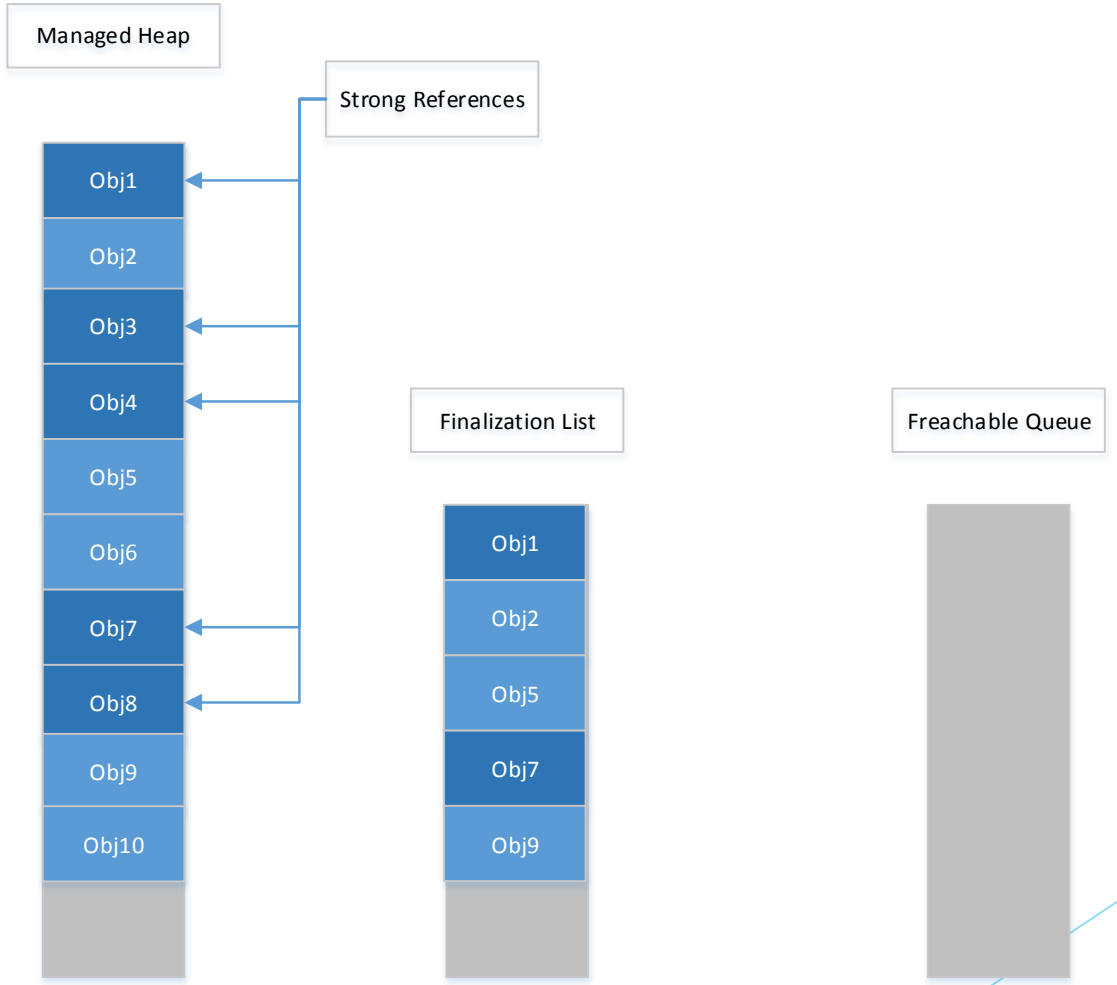
Пример механизма использования финализатора

- Шаг 1



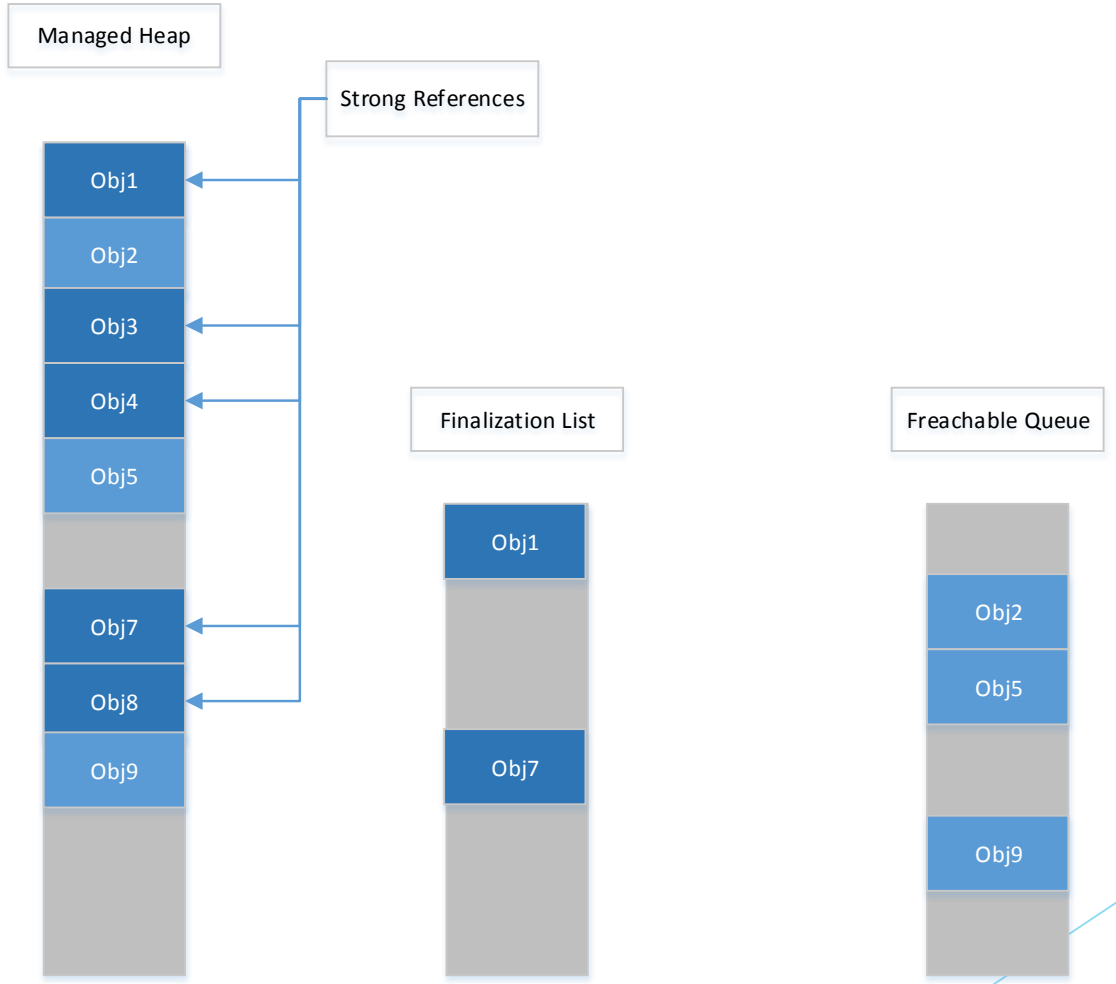
Пример механизма использования финализатора

- Шаг 2



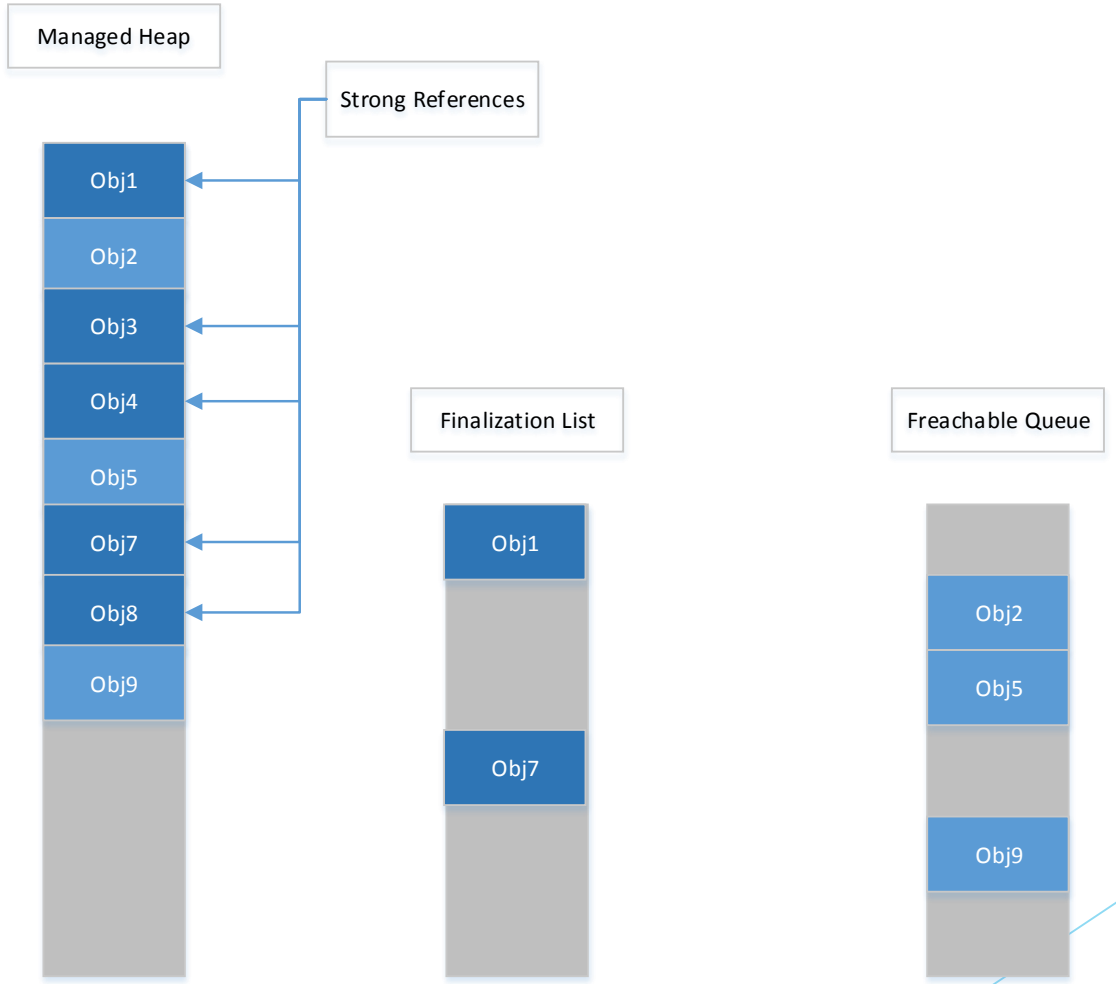
Пример механизма использования финализатора

- Шаг 3



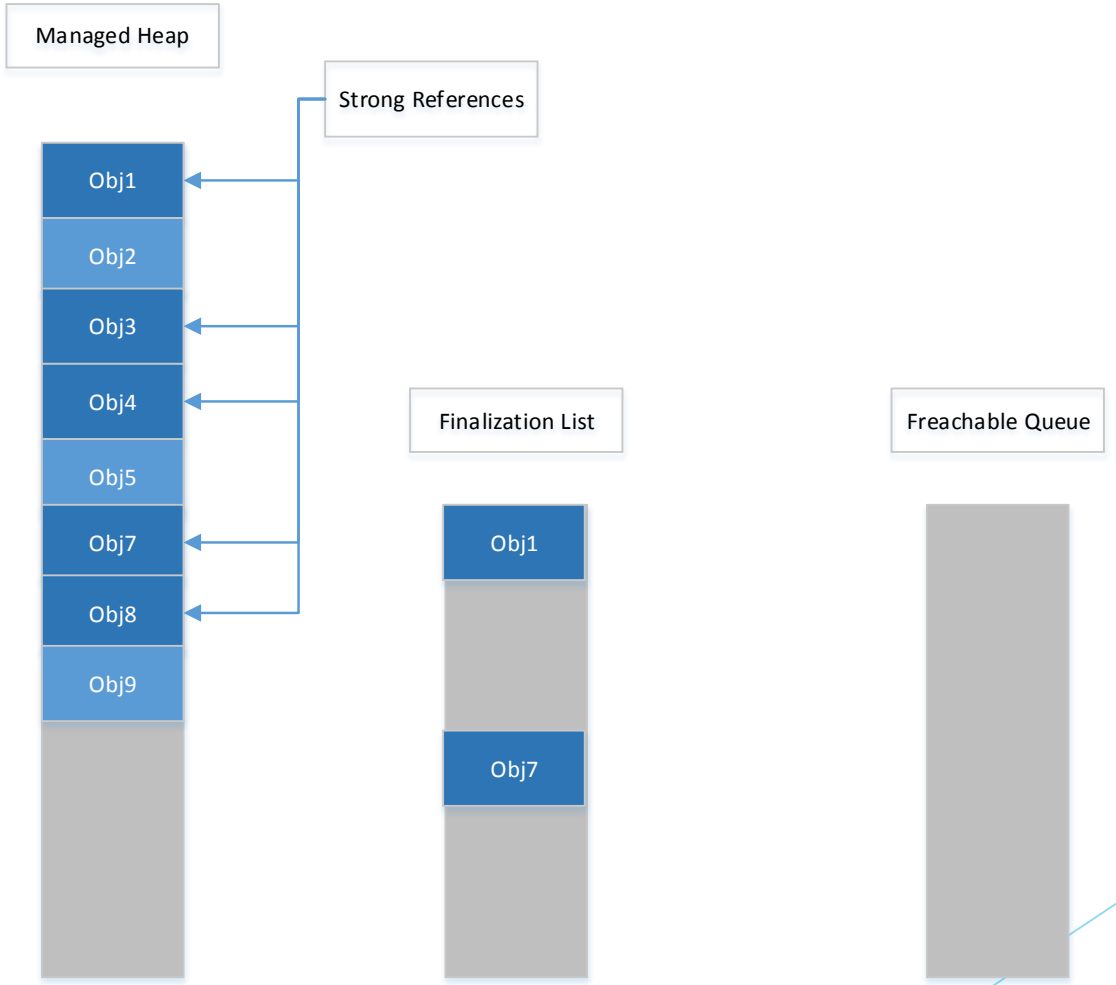
Пример механизма использования финализатора

- Шаг 4



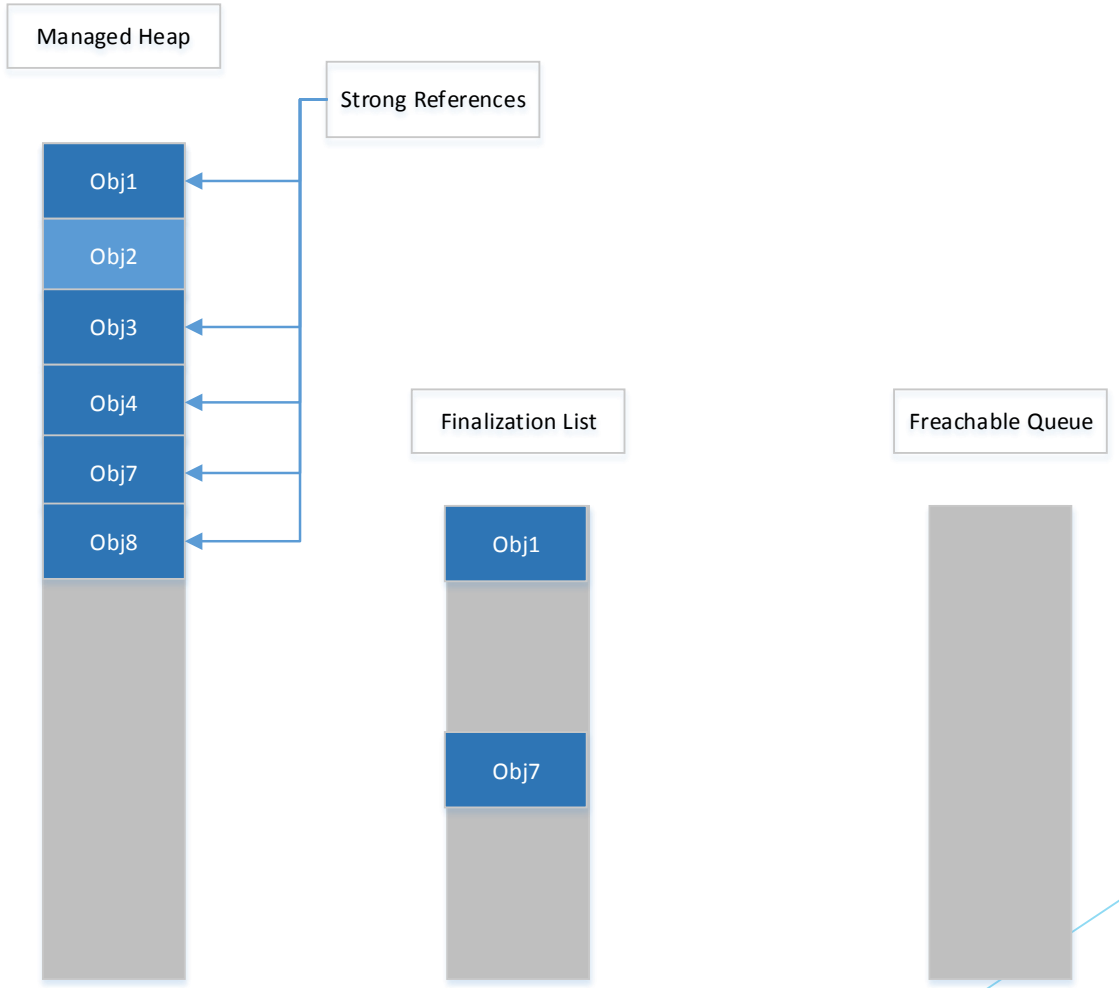
Пример механизма использования финализатора

- Шаг 5



Пример механизма использования финализатора

- Шаг 6



Управление ресурсами

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

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

Управление ресурсами в управляемой среде - пример

- 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

Управление ресурсами в управляемой среде –когда использовать?

- Разблокировка файлов
- Предотвращение потери кэшированных данных
- Контроль одновременных соединений с базой данных

Шаблон dispose

- Шаблон dispose является шаблоном проектирования, позволяющим высвободить неуправляемые ресурсы, используемые классом, контролируемо и своевременно. Реализация в типе этого шаблона будет способствовать тому, что приложения будут хорошо работать, и не сохранять неуправляемые ресурсы дольше, чем это необходимо. .NET Framework. предоставляя интерфейс IDisposable и объект, реализующий этот интерфейс, должен следовать этому паттерну.
- Для освобождения неуправляемых ресурсов могут применяться методы финализации при активизации процесса сборки мусора.
- 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 (метод Dispose)

Интерфейс IDispose

- 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.

Интерфейс IDiasposable в базовых классах

- 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.

- FileStream fs = new FileStream("File.txt", FileMode.OpenOrCreate);
- Fs.Close();
- Fs.Dispose();

Метод 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.

Использование метода Dispose

- 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.

Работа с управляемыми ресурсами

- 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. С
- борщик 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.
- Эта стратегия не гарантирует, что память, используемая массивом, будут возвращена раньше, чем могла бы. Время зависит от сборщика мусора.

Управление ресурсами в приложениях

- Существует несколько подходов, которые можно использовать, чтобы распоряжаться объектом, когда он больше не нужен:
  - Можно вручную вызвать метод Dispose на соответствующем шаге кода.
  - Можно использовать блок try/finally и ликвидировать объект в блоке finally.
  - Можно использовать блок using (не следует путать с ключевым словом using для импорта пространства имен) для инкапсуляции удаляемых объектов.
- Использование блока using 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.

Пример – управление ресурсами в приложениях

```
//1
LogFileWriter lfw = new LogFileWriter(...);
...
// Use the LogFileWriter object.
lfw.Dispose();
```

```
//2
LogFileWriter lfw;
try
{
    lfw = new LogFileWriter (...);
    ...
    // Use the LogFileWriter object.
    ...
}
finally
{
    if (lfw != null)
    {
        Lfw.Dispose();
    }
}
```

```
//3
using (LogFileWriter lfw = new LogFileWriter (...))
{
...
// Use the LogFileWriter object.
...
}
```

Домашнее задание

- Дополнить предыдущее задание следующим функционалом:
- Реализовать интерфейсы IClonable для копирования данных объекта, IComparable, а также перегрузить операторы для сравнения фигур и математических операций (с разными типами) и методы класса object