Тема 9. Создание и уничтожение объектов

Парадигмы программирования. C#

План

- Использование конструкторов
- Инициализация данных
- Объекты и память
- Управление ресурсами

Использование конструкторов

Часть 1

Создание объектов

- Шаг 1: Выделение памяти
  - Используется ключевое слово new для выделения памяти из кучи
- Шаг 2: Инициализация объекта с использованием конструктора
  - Используется имя класса со скобками
- Шаг 1 и шаг 2 нельзя выполнить один отдельно от другого

- Пример создания объекта

```
User user = new User();
```

- Альтернативные примеры создания объектов

```
string s = "Hello";
string s = new string(new
  char[]{'H','e','l','l','o'});
int[ ] array = {1, 2, 3, 4};
int[ ] array = new int[4]{1, 2, 3, 4};
```

Использование конструктора по умолчанию

- Для классов без конструктора C# автоматически создаёт конструктор по умолчанию
- Особенности конструктора по умолчанию
  - Публичная доступность
  - Имя, совпадающее с именем класса
  - Нет возвращаемого типа данных, даже не void
  - Нет аргументов
  - Инициализирует все поля в 0, false или null («нулевая» инициализация)
- Синтаксис конструктора

```
class User { public User() { … } }
```

Переопределение конструктора по умолчанию

- Если конструктор по умолчанию не подходит, создайте свой собственный конструктор
  - Нужен не публичный конструктор
  - «Нулевая» инициализация не всегда подходит
  - Невидимый код трудно обслуживать
- Пример переопределения конструктора по умолчанию

```
class User
{
    private string name;
    private DateTime birthday;

    public User()
    {
        name = "Vasya";
        birthday = DateTime.Now;
    }
}
```

Перегрузка конструкторов

- Конструкторы – тоже методы, поэтому их можно перегружать
  - Должны совпадать: область видимости и имя, должны отличаться только параметры
  - Перегруженные конструкторы позволяют инициализировать объекты разными способами
- Предупреждение
  - Если создадите конструктор для класса, компилятор не создаст конструктор по умолчанию

```
class User
{
    …

    public User(string name, DateTime birthday)
    {
        this.name = name; this.birthday = birthday;
    }
    public User()
    {
        name = "Vasya"; birthday = DateTime.Now;
    }
}
```

Инициализация данных

Часть 2

Списки инициализации

- Перегруженные конструкторы могут содержать дублирующийся код
  - Переработайте код для того, чтобы конструкторы вызывали друг друга
  - Используйте ключевое слово this в списках инициализации

```
class User
{
    …

    public User(string name, DateTime birthday)
    {
        this.name = name;
        this.birthday = birthday;
    }

    public User() : this("Vasya", DateTime.Now) { }
}
```

Ограничения списков инициализации

- Нельзя создавать списки инициализации конструктора, приводящие к вызову этого же конструктора

```
// Ошибка компиляции
public User(string name, DateTime birthday) :
    this(name, birthday) { }
```

- Нельзя использовать ключевое слово this в аргументах конструктора

```
class Point
{
 // Ошибка компиляции
    public Point() : this(X(this), Y(this)) { }
    public Point(int x, int y) { ... }
    private static int X(Point p) { ... }
    private static int Y(Point p) { ... }
}
```

Объявление неизменяемых полей и полей-констант

- Значение поля константы определяется во время компиляции

```
const int speedLimit = 90;
```

- Значение неизменяемого поля определяется во время выполнения

```
readonly int loopCount = 120;
```

Инициализация неизменяемых полей

- Неизменяемые поля обязательно должны быть инициализированы
  - Неявно в 0, false или null
  - Явно при объявлении в инициализаторе переменной
  - Явно внутри нестатического конструктора

```
class TextFile
{
    private readonly ArrayList lines;
}
```

Конструктор для структуры

- Компилятор
  - Компилятор всегда создаёт конструктор по умолчанию, даже если есть конструкторы не по умолчанию
  - Конструктор по умолчанию инициализирует все поля в нули
- Программист
  - Может создать конструкторы с одним или несколькими аргументами
  - Конструкторы, созданные вручную, не инициализируют поля в нули
  - Никогда не может создать конструктор по умолчанию
  - Внутри конструктора необходимо инициализировать все поля
  - Никогда не может создать конструктор с модификатором доступа protected

Использование частных конструкторов

- Частные конструкторы не позволяют создавать нежелательные объекты
  - Нельзя вызывать нестатические методы
  - Можно вызывать статические методы
  - Удобный способ реализации островка процедурного программирования
  - Частный конструктор используется в паттерне Singleton, позволяющем создавать класс от которого может быть получен только один объект

```
public class Math
{
    public static double Cos(double x) { ... }
    public static double Sin(double x) { ... }
    private Math() { }
}
```

Использование статических конструкторов

- Назначение
  - Вызываются загрузчиком класса во время выполнения программы
  - Могут использоваться для инициализации статических полей
  - Гарантированно вызываются перед вызовом нестатических конструкторов
- Ограничения
  - Не могут вызываться из кода
  - Не могут иметь модификатора доступа
  - Не могут иметь параметров
  - Не допускают использования ключевого слова this и списков инициализации

```
class Example
{
    static Example()
    {
        w = new Wibble();
    }
    private static Wibble w;
}
```

Объекты и память

Часть 3

Жизненный цикл объектов

- Создание объектов
  - Выделение памяти с помощью оператора new
  - Инициализация объекта в памяти с использованием конструктора
- Использование объектов
  - Вызов методов
- Уничтожение объектов
  - Объект возвращается обратно в память
  - Память деинициализируется

Объекты и область видимости

- Время жизни переменной значимого типа привязано к области, в которой она объявлена
  - Обычно короткое время жизни
  - Детерминированное создание и уничтожение
- Время жизни объекта не привязано к его области
  - Обычно время жизни дольше времени жизни локальных переменных
  - Недетерминированное уничтожение

Сборка мусора

- Явное удаление объектов не доступно
  - В C# нет оператора, противоположного оператору new (например, delete)
  - Причина: явное удаление объекта является одним из основных источников ошибок в других языках программирования
- Удалением объектов занимается сборщик мусора
  - Он находит недоступные объекты и уничтожает их
  - Он возвращает память, занятую уничтоженными объектами обратно в состав свободной памяти
  - Он обычно запускается, когда свободной памяти становится мало

Управление ресурсами

Часть 4

Очистка объектов

- Финальные действия различных объектов будут различными
  - Эти действия не могут определяться только сборщиком мусора
  - Все объекты в .NET Framework, начиная с класса Object, имеют метод Finalize
  - Напрямую вызывать метод Finalize или переопределять его нельзя
  - Деструктор – это то же самое, что и метод Finalize, но его уже можно переопределить
  - В C# для написания кода очистки данных используются деструкторы
  - При наличии деструктора он будет вызван сборщиком мусора перед очисткой памяти

Написание деструкторов

- Деструктор – это механизм очистки
- Синтаксис деструкторов
  - Отсутствует модификатор доступа
  - Отсутствует возвращаемый тип данных, даже void
  - Имя совпадает с именем класса, перед которым стоит ~
  - Не имеет параметров

```
class TextFile
{
    ~TextFile( ) { ... }
}
```

Предупреждения о времени работы деструктора

- Порядок время деструкции не определено
  - Необязательна полная противоположность конструктору
- Гарантируется, что деструктор будет вызван
  - Нельзя рассчитывать на время вызова деструктора
- Старайтесь по возможности избегать деструкторов
  - Они снижают производительность
  - Они усложняют программу
  - Они приводят к задержке при высвобождении ресурсов

Интерфейс IDisposable

- Действия по очистке ресурса
  - Унаследуйте класс от интерфейса IDisposable и реализуйте метод Dispose, который будет очищать ресурсы
  - Внутри метода Dispose вызовите метод GC.SuppressFinalize, что исключит вызов деструктора для данного объекта
  - Убедитесь, что при работе с объектом метод Dispose вызывается только один раз
  - Убедитесь, что объект после вызова Dispose больше не используется

Пример реализации интерфейса IDisposable

- http://msdn.microsoft.com/en-us/library/system.idisposable.aspx

```
using System;
using System.ComponentModel;

public class DisposeExample
{
    public class MyResource : IDisposable
    {
        private IntPtr handle;
        private Component component = new Component();
        private bool disposed = false;

        public MyResource(IntPtr handle)
        {
            this.handle = handle;
        }
```

Пример реализации интерфейса IDisposable (продолжение)

```
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

private void Dispose(bool disposing)
{
    if (!this.disposed)
    {
        if (disposing)
        {
            component.Dispose();
        }

        CloseHandle(handle);
        handle = IntPtr.Zero;

        disposed = true;
    }
}
```

Пример реализации интерфейса IDisposable (окончание)

```
        [System.Runtime.InteropServices.DllImport("Kernel32")]
        private extern static Boolean CloseHandle(IntPtr handle);

        ~MyResource()
        {
            Dispose(false);
        }
    }
    public static void Main()
    {
        MyResource myResource = new MyResource(IntPtr.Zero);
        myResource.Dispose();
    }
}
```

Оператор using

- Синтаксис

```
using (Resource r = new Resource())
{
    r.Method();
}
```

- Метод Dispose автоматически вызывается в конце блока using
- Эквивалентный пример

```
Resource r = new Resource();
try
{
    r.Test();
}
finally
{
    if (r != null) ((IDisposable)r).Dispose();
}
```

Заключение

- Спасибо за внимание!