Лекция 8

Интерфейсы

- ICloneable
- IComparable и IComparer
- IEnumerator и IEnumirable
- IDisposible

ICloneable

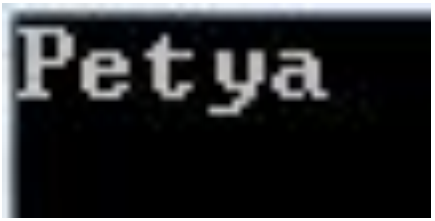
- Поддерживает копирование, который создает **новый экземпляр** класса с тем же значением, что существующий экземпляр.

- Перегружаемый метод: Clone();

Зачем он нужен?

- Пример:

```
public class ClassUser
{
    1 reference
    public string Login { set; get; }

    1 reference
    public string Password { set; get; }
}
```



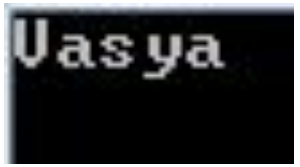

```
static void Main(string[] args)
{
    ClassUser cl1 = new ClassUser() { Login = "Vasya", Password = "Pupkin" };
    ClassUser cl2 = cl1;
    cl2.Login = "Petya";
    Console.WriteLine(cl1.Login);
    Console.ReadKey();
}
```

Зачем он нужен?

- В данном случае объекты cl1 и cl2 будут указывать на **один и тот же объект** в памяти, поэтому изменения свойств в переменной cl2 затронут также и переменную cl1.
- Чтобы переменная cl2 указывала на новый объект, но со значениями из cl1, его необходимо **клонировать**.

Реализация

```
public class ClassUser : ICloneable
{
    5 references
    public string Login { set; get; }

    3 references
    public string Password { set; get; }

    0 references
    public object Clone()
    {
        return new ClassUser() { Login = this.Login, Password = this.Password };
    }
}
```




```
static void Main(string[] args)
{
    ClassUser cl1 = new ClassUser() { Login = "Vasya", Password = "Pupkin" };
    ClassUser cl2 = (ClassUser)cl1.Clone();
    cl2.Login = "Petya";
    Console.WriteLine(cl1.Login);
    Console.ReadKey();
}
```

Можно проще

- Для сокращения кода копирования мы можем использовать специальный метод **MemberwiseClone()**, который возвращает копию объекта

```
public class ClassUser : ICloneable
{
    3 references
    public string Login { set; get; }

    1 reference
    public string Password { set; get; }

    1 reference
    public object Clone()
    {
        return MemberwiseClone();
    }
}
```

Недостаток

- Этот метод реализует **поверхностное (неглубокое) копирование**.

- Если в классе есть поля-объекты от других классов, то в объекте-клоне создастся не новый объект, а копируется ссылка от текущего

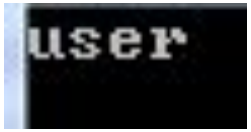
Добавим класс

```
public class Role
{
    0 references
    public string RoleName { set; get; }
}
```

```
public class ClassUser : ICloneable
{
    3 references
    public string Login { set; get; }

    1 reference
    public string Password { set; get; }

    0 references
    public Role UserRole { set; get; }

    1 reference
    public object Clone()
    {
        return MemberwiseClone();
    }
}
```

Что получаем

```
static void Main(string[] args)
{
    ClassUser cl1 = new ClassUser()
    {
        Login = "Vasya",
        Password = "Pupkin",
        UserRole = new Role() { RoleName = "admin" }
    };
    ClassUser cl2 = (ClassUser)cl1.Clone();
    cl2.UserRole.RoleName = "user";
    Console.WriteLine(cl1.UserRole.RoleName);
    Console.ReadKey();
}
```



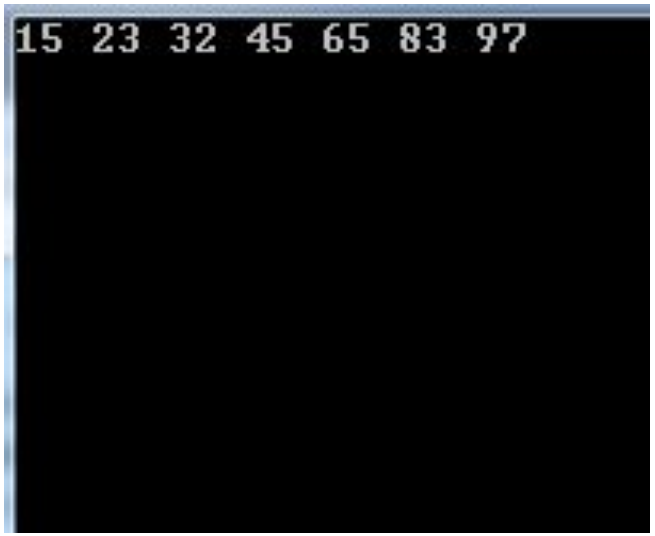
Глубокое копирование

```
public object Clone()
{
    return new ClassUser()
    {
        Login = this.Login,
        Password = this.Password,
        UserRole = new Role() { RoleName = this.UserRole.RoleName }
    };
}
```

```
public object Clone()
{
    return new ClassUser()
    {
        Login = this.Login,
        Password = this.Password,
        UserRole = (Role)this.UserRole.Clone()
    };
}
```

IComparable

```
static void Main(string[] args)
{
    int[] numbers = new int[] { 97, 45, 32, 65, 83, 23, 15 };
    Array.Sort(numbers);
    foreach (int n in numbers)
    {
        Console.Write(n + " ");
    }
    Console.ReadKey();
}
```



IComparable

- Однако метод Sort по умолчанию работает только для наборов примитивных типов, как int или string. Для сортировки наборов сложных объектов применяется интерфейс **IComparable**.

```
int CompareTo(object o);
```

Возвращаемое значение

- **Меньше нуля**. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра
- **Равен нулю**. Значит, оба объекта равны
- **Больше нуля**. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра

Реализация

```
public class ClassUser : ICloneable, IComparable
{
    4 references
    public string Login { set; get; }

    2 references
    public string Password { set; get; }

    2 references
    public Role UserRole { set; get; }

    2 references
    public object Clone()...

    0 references
    public int CompareTo(object obj)
    {
        if (obj is ClassUser)
        {
            var user = obj as ClassUser;
            return this.Login.CompareTo(user.Login);
        }
        else
            throw new Exception("Невозможно сравнить два объекта");
    }
}
```

Еще вариант

```
public int CompareTo(object obj)
{
    if (obj is ClassUser)
    {
        var user = obj as ClassUser;
        var res = this.UserRole.RoleName.CompareTo(user.UserRole.RoleName);
        if (res != 0)
        {
            return res;
        }
        return this.Login.CompareTo(user.Login);
    }
    else
        throw new Exception("Невозможно сравнить два объекта");
}
```

IComparer

Кроме интерфейса IComparable имеется интерфейс IComparer

int Compare(object o1, object o2);

- Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения:

IComparer

- если первый объект больше второго, то возвращается число больше о,
- если меньше - то число меньше нуля;
- если оба объекта равны, возвращается ноль.

Реализация

```
public class Role : ICloneable, IComparer<Role>
{
    6 references
    public string RoleName { set; get; }

    2 references
    public object Clone()...

    0 references
    public int Compare(Role x, Role y)
    {
        if (x.RoleName.Length > y.RoleName.Length)
            return 1;
        else if (x.RoleName.Length < y.RoleName.Length)
            return -1;
        else
            return 0;
    }
}
```

IEnumerable

- Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс - перечислитель:

```
public interface IEnumerable
{
   IEnumerator GetEnumerator();
}
```

IEnumerator

- Интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере.

IEnumerator

```
public interface IEnumerator
{
   bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов
   object Current {get;}  // текущий элемент в контейнере
   void Reset(); // перемещение в начало контейнера
}
```

Реализация

```
public class RoleList : IEnumerator<Role>
{
    private List<Role> roles = new List<Role>()
    {
        new Role() { RoleName = "Role1" } ,
        new Role() { RoleName = "Role2" } ,
        new Role() { RoleName = "Role3" } ,
        new Role() { RoleName = "Role4" } ,
        new Role() { RoleName = "Role5" } ,
        new Role() { RoleName = "Role6" }
    };

    int currentIndex = -1;
```

```
public Role Current
{
    get { return roles[currentIndex]; }
}

0 references
public bool MoveNext()
{
    if(currentIndex + 1 == roles.Count)
    {
        Reset();
        return false;
    }
    currentIndex++;
    return true;
}

1 reference
public void Reset()
{
    currentIndex = -1;
}
```

Реализация

```
public class RoleEnum : IEnumerable<Role>
{
    RoleList rl = new RoleList();
    0 references
    public IEnumerator<Role> GetEnumerator()
    {
        return rl;
    }

    0 references
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }
}
```

Применение

```
static void Main(string[] args)
{
    RoleEnum re = new RoleEnum();
    foreach(var elem in re)
    {
        Console.WriteLine("Role {0}", elem.RoleName);
    }
    Console.ReadKey();
}
```

```
Role Role1
Role Role2
Role Role3
Role Role4
Role Role5
Role Role6
```

IDispolable

- Предоставляет механизм для освобождения управляемых и неуправляемых ресурсов.

void Dispose()

Ресурсы

- Существуют два различных подхода, которые можно применять для создания класса, способного производить очистку и освобождать внутренние неуправляемые ресурсы.

Ресурсы

- Первый подход заключается в переопределении метода System.Object.Finalize() и позволяет гарантировать то, что объект будет очищать себя сам во время процесса сборки мусора (когда бы тот не запускался) без вмешательства со стороны пользователя.

Ресурсы

- торой 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 оставаться в памяти на неопределенный срок.

Ресурсы

- Если пользователь объекта не забыл вызвать метод Dispose(), можно проинформировать сборщик мусора о пропуске финализации, вызвав метод GC.SuppressFinalize().
- Если же пользователь забыл вызвать этот метод, объект рано или поздно будет подвергнут финализации и получит возможность освободить внутренние ресурсы.

Виды ресурсов

- Неуправляемые ресурсы - это разнообразные файловые хэндлы, оконные, всякие объекты синхронизации, соединения с базой данных
- Управляемые ресурсы - всё, что создаётся средствами .net, и все объекты

Сборщик мусора

- Сборщик мусора автоматически освобождает память, выделенную для управляемого объекта, если этот объект больше не используется.
- Сборщик мусора не имеет сведений о неуправляемых ресурсов, таких как дескрипторы окон, или открытые файлы и потоки.

IDispolable

- Использование метода Dispose, позволяет явно освобождать неуправляемые ресурсы вместе со сборщиком мусора. Пользователь объекта может вызвать этот метод, когда объект больше не нужен.

IDispolable

```
public class RoleList : IEnumerator<Role>, IDisposable
{
    bool disposed = false;

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

    0 references
    public ~RoleList()
    {
        Dispose(false);
    }

    2 references
    protected virtual void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                //отчистка управляемых ресурсов
            }
            //отчистка неуправляемых ресурсов
            this.disposed = true;
        }
    }
```

IDispolable

- Если есть подключение к файлу или к БД, которое «живет» на всем протяжении работы объекта класса, то в Displose в секторе удаления управляемой памяти нужно вызывать методы Dispose этих классов.

Использование

- Есть 2 варианта как правильно реализовывать отчистку
  - Либо через using
  - Либо напрямую вызывать Dispose()