# IComparer

Кроме интерфейса IComparable имеется интерфейс Icomparer (компаратор) – сравнение по различным критериям – объявлен в пространстве имён System.Collection:

**interface IComparer { int Compare(object ob1, object ob2)}**

Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения:
int Compare(object o1, object o2);

# IComparer

- если первый объект больше второго, то возвращается число больше 0,
- 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

- 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 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 очистку:
  - либо через using;
  - либо напрямую вызывать Dispose().