C# Перечисления

***Перечисления* –** позволяют создать набор констант. Это типы со значениями.

Синтаксис –

enum enum_name {contant_list}

enum_name – имя перечисления (Первая буква – заглавная)

contant_list – список констант входящих в перечисление

Enum Planet { Mercury, Venus, Earth, Mars, Jupiter, Saturn}

По умолчанию первая константа имеет значение 0, вторая – 1 и т.д.

Это можно изменить

Enum Planet { Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn}

C# **перечисления**

Доступ к элементам перечислений через оператор точка

System.Console.WriteLine(“ Земля это планета номер “ + (int)Planet.Earth);

Здесь привели тип элемента перечисления к типу int.

Если этого не сделать то Planet.Earth вернуло бы строку “Earth”.

Можно указывать значения констант

```
Enum PlanetPeriods
{
Mercury = 88,
Venus = 225,
Earth = 365
}
```

C# **перечисления**

```
System.Console.WriteLine(“Период обращения для
    Меркурия =  “ + (int)PlanetPeriods.Mercury + “дней”);
```

В перечислениях можно указывать базовый тип

```
Enum PlanetPeriods:long
{
Mercury = 88,
Venus = 225,
Earth = 365
}
```

C# оператор **is**

Этот оператор определяет принадлежность переменной. константы, выражения заданному типу

```
int Myint = 0;
bool compatible = Myint is int; // здесь возвращается true
bool compatible = Myint is double; // здесь возвращается false
```

Этот оператор используют тогда, когда нужно проверить, унаследован ли тип объекта от заданного класса или реализует ли объект заданный интерфейс.

C# оператор **switch**

Такой же как в C++, но может работать и для строковых значений

```
 string planet = “Earth”;
 switch( planet)
{
    case “Mercury” :
     System.Console.WriteLine(1);
     break;
    case “Venus” :
     System.Console.WriteLine(2);
     break;
```

C# **switch**

```
    case “Earth” :
     System.Console.WriteLine(3);
     break;
    default :
     System.Console.WriteLine(“Unknown planet”);
     break;
}
```

C# цикл **foreach**

Этот цикл позволяет проходить по набору элементов

Синтаксис

```
 foreach (тип имя_переменной in выражение)
{
    тело цикла
}
 int [ ] myVal = {2, 4, 3, 5, 1};
 foreach( int count in myVal)
{
    System.Console.WriteLine(“ counter = “ + counter);
}
```

C# модификаторы уровня доступа в классах

public – компонент доступен без ограничений

protected internal - компонент доступен только внутри класса, из производного класса или из класса в той же программе

internal - компонент доступен только внутри класса, или из класса в той же программе

protected - компонент доступен только внутри класса, или из производного класса

private - компонент доступен только внутри класса

C# **создание и уничтожение объектов**

Объекты создаются с помощью конструкторов

```
Car myCar = new Car();
```

Переменные, объекты и строки уничтожаются в ходе процесса **сборки мусора.** Сборщик мусора периодически вычищает из памяти данные, которые больше не используются. Когда переменная выходит из области видимости она автоматически ставится в очередь на уничтожение сборщиком мусора.

В принципе можно не заботится об очистке памяти от ненужных переменных, однако все таки иногда это надо делать. Например, если объект открывает файл, то перед уничтожением объекта надо этот файл корректно закрывать. Для этого надо использовать деструкторы.

C# **создание и уничтожение объектов**

В деструкторе надо выполнить все необходимые действия кроме очистки памяти.

Деструктор вызывается автоматически перед постановкой объекта в очередь на уничтожение.

```
public class Car
{
   ~Car()
   {
    System.Console.WriteLine(“We are in destructor ~Car()”);
   }
}
  class Test
{
```

C# **создание и уничтожение объектов**

```
public static void Main()
{
  Car myCar = new Car();
  System.Console.WriteLine(“We are in the end of Main()”);
}
```

Результат запуска

We are in the end of Main()”

We are in destructor ~Car()

Деструктор вызвался в самом конце метода Main() !!!

C# **свойства (property)**

Их назначение – получать и устанавливать значения полей при помощи методов. Это позволяет скрыть поля от пользователей, сделав их private, но сохранить к ним привычный, но **контролируемый доступ**.

В свойствах могут определяться 2 метода - **set** и **get** (при их объявлении круглые скобки опускаются)

Метод get – возвращает значение поля, set его устанавливает

```
public class Car
{
    private string model;
    public string Model
```

C# **свойства (property)**

```
    get
    {
      return model;
    }
    set
    {
      model = value;
    }
}

Car myCar = new Car();
myCar.Model = “Toyota”;
SystemConsole.WriteLine(“myCar.Model = “ + myCar.Model);
```

C# **Пространство имен**

Пространства имен ограничивают объявления классов определенным участком кода.

```
 namespase Shevrolet
{
    public class Car
    {
      public string model;
    }
}

 namespase Bmw
{
    public class Car
    {
      public string model;
    }
}
```

C# **Пространство имен**

Объявление класса Car в разных пространствах не конфликтуют друг с другом. Однако можно использовать классы из другого пространства имен даже с одинаковыми названиями.

Chevrolet.Car myCar = new Chevrolet.Car();

При этом необходимо указывать имя пространства через точку перед именем класса.

Пространства имен можно вкладывать одно в другое, получая иерархию имен. Это нужно для того, чтобы можно было вести независимую разработку частей проекта разными командами.

```
namespase Shevrolet
{
    namespace UserInterface
    {
      // классы
    }
    namespace DataBaseAcces
    {
      // классы
    }
}
```

C# **Пространство имен**

Оба пространства вложены в пространство имен Shevrolet.

Вложенные пространства могут в свою очередь содержать другие вложенные пространства имен

```
 namespase Shevrolet
{
    namespace UserInterface
    {
     namespace WelcomeScreen
     {
          // классы
     }
    }
}
```

C# **Пространство имен**

Пространства можно разделять точкой

```
namespase Shevrolet.UserInterface
{
    // классы
}
namespase Shevrolet.DataBaseAccess
{
    // классы
}
```

Иерархия пространств иен может быть разбросана по разным файлам. При этом каждая команда программистов работает только со своими файлами.

C# **Пространство имен**

```
    namespase Shevrolet
{
    namespace UserInterface
    {
     public class MyClass
     {
          public void Test()
          {
               System.Console.WriteLine(“Interface Test()”);
          }
     }
    }
}
```

C# **Пространство имен**

```
namespase Shevrolet.DataBaseAccess
{
    namespace UserInterface
    {
     public class MyClass
     {
          public void Test()
          {
               System.Console.WriteLine(“DataBase Test()”);
          }
     }
    }
}
```

C# **Пространство имен**

```
    class Example
{
    public void Main()
    {
     Shevrolet.UserInterface.MyClass  myUI =
          new       Shevrolet.UserInterface.MyClass();
     Shevrolet.DataBaseAccess.MyClass  myDB =
          new       Shevrolet.DataBaseAccess.MyClass();
     myUi.Test();
     myDB.Test();
    }
}
```

После запуска на экране будет

Interface Test()

DataBase Test()

C# **Пространство имен**

Когда имеется большая вложенность пространства имен удобно использовать оператор **using**

```
 using System;
…..
 class MyClass
{
    Console.WriteLine(“ Hello”);
}
```

Указать пространство имен можно до того как оно объявлено

```
 using Bmw
 namespace Bmw
{
    …..
}
```

C# Пространство имен

```
using Bmw;
using System;
namespace Bmw
{
  public class Car
  {
        public string model;
  }
}
class Example
{
  public static void Main()
  {
    Console.WriteLine(“Create object Bmw.Car”);
    Car MyCar = new Car();
    MyCar.model = “ Bmw X5”;
```

C# **Пространство имен**

```
  Console.WriteLine("MyCar.model = "  + mayCar.model);
    }
}
```

На экране будет

Create object Bmw.Car

MyCar.model = Bmw X5

C# **Интерфейсы**

Интерфейс – список объявлений методов и свойств. Созданный интерфейс может быть реализован в некотором классе. Этот класс должен содержать код, соответствующий объявлениям в интерфейсе.

Класс, реализующий интерфейс, гарантирует, что для всех элементов, объявленных в интерфейсе, представлен код. Класс может реализовывать несколько интерфейсов.

Объявление интерфейса – синтаксис

[модификатор_уровня_доступа] interface имя_интерфейса

{

тело_интерфейса

}

модификатор_уровня_доступа – только **public** и **internal**

имя_интерфейса – по соглашению должно начинаться с буквы **I**

тело_интерфейса – объявляются методы, свойства, индексаторы но не поля !!!

C# **Интерфейсы**

```
 public interface IDrivable
{
    // методы
    void Start();
    void Stop();
    // свойство
    bool Started
    {
      get;
    }
}
```

Обявления в интерфейсе не содержат кода. Код реализуется в классе, поддерживающий интерфейс.

C# **Интерфейсы**

```
public class Car : IDrivable // формализм аналогичен наследованию
{
    private bool started = false;
    // реализация методов
    public void Start()
    {
      Console.WriteLine("Машина завелась");
      started = true;
    }
    public void Stop()
    {
      Console.WriteLine("Машина остановилась");
      started = false;
    }
    // реализация свойства
    public bool Started
    {
```

.

C# **Интерфейсы**

```
      get
      {
            return started;
      }
    }
}

class Example
{
    public static Main()
    {
      Car myCar  = new Car();
      myCar.Start();
      Console.WriteLine(“myCar.Started = “ + myCar.Started);
      myCar.Stop();
      Console.WriteLine(“myCar.Started = “ + myCar.Started);
```

C# **Интерфейсы**

```
    }
}
```

При запуске на экране будет

Машина завелась

myCar.Started = True

Машина остановилась

myCar.Started = False