### Интернирование строк

class Program

{

static void Main(string[] args)

{

string s1 = "str";

string s2 = "str";

Console.WriteLine(Object.ReferenceEquals(s1, s2));

Console.ReadLine();

}

}

#### Использование Using

* namespace
* IDisposable
* Псевдоним Using I = System.Unt32; (MyInt i = 0;)

#### Инкапсуляция

* Сокрытие данных
* Предотвращение вмешательства в работу объекта

#### Наследование

* Процесс наследования методов и свойств родительских классов

#### Полиморфизм

* Возможность переопределения поведения в наследуемых сущностях

#### Классы

[модификаторы] [ключевое слово класс][ название класса][ ограничения универсального типа]

* public: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
* private: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
* protected: такой член класса доступен из любого места в текущем классе или в производных классах.
* internal: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
* protected internal: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
* sealed – запрещает другим классам наследовать от этого класса
* static - объявление статического члена, принадлежащего собственно типу, а не конкретному объекту
* abstract - класс предназначен только для использования в качестве базового класса для других классов

#### Автосвойства

public string Field {get; set;}

//Эквивалент

public class C

{

private string \_Field;

public string Field

{

get

{

return \_Field;

}

set

{

\_Field = value;

}

}

//C# 6

public string FirstName { get; set; } = "Hello";

}

#### Конструктор класса

Конструктор-это специальный метод, который вызывается при создании экземпляра объекта.

В .Net Framework,если в созданном вами классе нет конструктора, описанного вами, то

создаётся конструктор без параметров, который заполняет поля объекта данными по умолчанию (0,false,null).

Если мы хотим, чтобы при создании объекта объект был заполнен какими-то данными, то

мы создаём конструктор с параметрами.

public class Person

{

public Person(string name, int age)

: this(name) //вызов приватного конструктора вместе с обычным конструктором

{

}

private Person(string name)

{

}

}

#### Использование статических конструкторов

class ScopeMonitor

{

static string urlFragment = "foo/bar";

static string firstPart= "http://www.example.com/";

static string fullUrl= firstPart + urlFragment;

}

class ScopeMonitor

{

static string firstPart= "http://www.example.com/";

static string fullUrl;

static string urlFragment = "foo/bar";

static ScopeMonitor()

{

fullUrl= firstPart + urlFragment;

}

}

### Синглтон

class Singleton

{

private static Singleton \_instance;

protected Singleton()

{

}

public static Singleton Instance()

{

if (\_instance == null)

{

\_instance = new Singleton();

}

return \_instance;

}

}

#### Private constructor

public class MyClass

{

private MyClass(object data1, string data2) { }

public MyClass(object data1) : this(data1, null) { }

public MyClass(string data2) : this(null, data2) { }

public MyClass() : this(null, null) { }

}

***Вложенные классы***

public class BaseClass

{

private BaseClass() { }

public class SubClass1 : BaseClass

{

public SubClass1() : base() { }

}

public class SubClass2 : BaseClass

{

public SubClass2() : base() { }

}

}

#### Перегрузка операторов

Overloading

Наличие методов с одинаковым название и с различной сигнатурой (входными параметрами)

class Test

{

static void Foo(int x)

{

Console.WriteLine("Foo(int x)");

}

static void Foo(string y)

{

Console.WriteLine("Foo(string y)");

}

static void Main()

{

Foo("text");

Foo(1);

}

}

#### Структуры

public struct CoOrds

{

public int x, y;

public CoOrds(int p1, int p2)

{

x = p1;

y = p2;

}

}

#### Переопределение +, ToString

public struct Complex

{

public int real;

public int imaginary;

// Constructor.

public Complex(int real, int imaginary)

{

this.real = real;

this.imaginary = imaginary;

}

public static Complex operator +(Complex c1, Complex c2)

{

return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);

}

public override string ToString()

{

return (System.String.Format("{0} + {1}i", real, imaginary));

}

}

class TestComplex

{

static void Main()

{

Complex num1 = new Complex(2, 3);

Complex num2 = new Complex(3, 4);

Complex sum = num1 + num2;

System.Console.WriteLine("First complex number: {0}", num1);

System.Console.WriteLine("Second complex number: {0}", num2);

System.Console.WriteLine("The sum of the two numbers: {0}", sum);

System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

#### Агрегация + композиция

class CompositeCustomService

{

// Композиция

private readonly CustomRepository \_repository = new CustomRepository();

public void DoSomething()

{

// Используем \_repository

}

}

class AggregatedCustomService

{

// Агрегация

private readonly AbstractRepository \_repository;

public AggregatedCustomService(AbstractRepository repository)

{

\_repository = repository;

}

public void DoSomething()

{

// Используем \_repository

}

}

#### Виртуальный метод

class TestOverride

{

public class Employee

{

public string name;

protected decimal basepay;

public Employee(string name, decimal basepay)

{

this.name = name;

this.basepay = basepay;

}

public virtual decimal CalculatePay()

{

return basepay;

}

}

public class SalesEmployee : Employee

{

private decimal salesbonus;

public SalesEmployee(string name, decimal basepay,

decimal salesbonus) : base(name, basepay)

{

this.salesbonus = salesbonus;

}

public override decimal CalculatePay()

{

return basepay + salesbonus;

}

}

static void Main()

{

// Create some new employees.

SalesEmployee employee1 = new SalesEmployee("Alice",

1000, 500);

Employee employee2 = new Employee("Bob", 1200);

Console.WriteLine("Employee4 " + employee1.name + " earned: " + employee1.CalculatePay());

Console.WriteLine("Employee4 " + employee2.name + " earned: " + employee2.CalculatePay());

Console.WriteLine($"Employee4 {employee1.name} earned: {employee1.CalculatePay()}");

Console.WriteLine($"Employee4 {employee2.name} earned: {employee2.CalculatePay()}");

}

}

#### Override vs new

class BaseClass

{

public virtual void Method1()

{

Console.WriteLine("Base - Method1");

}

public virtual void Method2()

{

Console.WriteLine("Base - Method2");

}

}

class DerivedClass : BaseClass

{

public override void Method1()

{

Console.WriteLine("Derived - Method1");

}

public new void Method2()

{

Console.WriteLine("Derived - Method2");

}

}

class Program

{

static void Main(string[] args)

{

BaseClass bc = new BaseClass();

DerivedClass dc = new DerivedClass();

BaseClass bcdc = new DerivedClass();

bc.Method1();

bc.Method2();

dc.Method1();

dc.Method2();

bcdc.Method1();

bcdc.Method2();

}

}