МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

ПРАКТИКУМ

по Microsoft .NET и языку программирования C#

Учебно-методическое пособие

Часть 1. Основы программирования на C#

2018 г.

УДК 681.3.06
П 56

Вл. Пономарев. Практикум по Microsoft .NET и языку программирования C#. Учебно-методическое пособие. Часть 1. Основы программирования на C#. Озерск: ОТИ НИЯУ МИФИ, 2018. — 50 с.

В пособии предлагаются практические работы по изучению платформы Microsoft .NET и языка программирования C#.

В первой части работ описываются основы программирования на C#.

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1. Синяков В. Е., начальник УИТ ФГУП «ПО «Маяк».
2. Зубаиров А. Ф., ст. преподаватель кафедры ПМ ОТИ НИЯУ МИФИ.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Общие цели занятий 5
1. Работа CS-101. Введение в C# 6
1.1. Консольное приложение Hello 6
1.2. Чтение и запись в консоль 8
1.3. Массивы 9
1.4. Строки 10
1.5. Перечисления 12
1.6. Структурные и ссылочные типы 13
1.7. Все есть объект 14
1.8. Случайные числа 16
1.9. Статические методы класса Environment 16
1.10. Контрольные вопросы и упражнения 17
2. Работа CS-102. Классы в C# 18
2.1. Определение структуры 18
2.2. Определение класса 19
2.3. Внутренние классы 20
2.4. Свойства в классах 21
2.5. Статический конструктор 22
2.6. Наследование и полиморфизм 22
2.7. Пространства имен 25
2.8. Контрольные вопросы и упражнения 26
3. Работа CS-103. Индексаторы 27
3.1. Простой индексатор 27
3.2. Составной индексатор 28
3.3. Контрольные вопросы и упражнения 28
4. Работа CS-104. Исключения 29
4.1. Генерация и перехват исключений 29
4.2. Пользовательские исключения 30
4.3. Делегирование методов 31
4.4. Контрольные вопросы и упражнения 33
5. Работа CS-105. Абстрактные классы и интерфейсы 34
5.1. Базовый класс 34
5.2. Абстрактный класс 35
5.3. Абстрактный метод 36
5.4. Интерфейсы 36
5.5. Определение наличия интерфейса 37
5.6. Интерфейсы как параметры методов 39
5.7. Иерархия интерфейсов 39
5.8. Контрольные вопросы и упражнения 40
6. Работа CS-106. Интерфейсы и коллекции C# 41
6.1. Интерфейсы перечисления 41
6.2. Внутренний перечислитель 43

6.3. Клонируемые объекты 45
6.4. Сравниваемые объекты 46
6.5. Коллекции C# 48
6.6. Контрольные вопросы и упражнения 50

Общие цели занятий

В ходе практических работ изучаются основы использования классов в рамках методологии объектно-ориентированного программирования.

В этой части работ рассматриваются следующие темы:

- аргументы командной строки;
- ввод и вывод в консоль;
- строки и массивы;
- структуры и классы;
- индексаторы;
- обработка исключений;
- наследование и полиморфизм;
- пространства имен;
- интерфейсы и иерархии интерфейсов;
- интерфейсы и коллекции C#.

К практическим работам приписаны контрольные вопросы и упражнения. Контрольные вопросы могут быть заданы преподавателем в ходе защиты работы, однако преподаватель может задавать и другие вопросы, не указанные в списке.

Для защиты знаний и навыков, полученных в ходе выполнения практических работ студент должен иметь тетрадь (12-18 листов). Для каждой выполненной работы в тетрадь записывается отчет.

Отчет по работе начинается с заголовка:

1. Фамилия Имя Отчество
2. Группа
3. Дата начала выполнения работы
4. Код работы
5. Название работы
6. Цели работы
7. Задачи работы

При необходимости при выполнении работы в отчет записываются контрольные значения. Во время защиты тетрадь используется для вопросов преподавателя и ответов студента.

Примеры программ данного сборника работ заимствованы из книги «Троэлсен Э. C# и платформа .NET. СПб.: Питер, 2005. — 796 с.: ил.».

1. Работа CS-101. Введение в C#

Цели:
- введение в консольные приложения C#.
Задачи:
- базовые сведения о программировании в среде .NET;
- ввод и вывод в консоль;
- строки и массивы;
- структуры и классы;
- сведения о системе.

1.1. Консольное приложение Hello

Главная особенность при программировании на C# — сначала создается класс, в котором определен статический метод Main с параметром или без него. Код программы описывается внутри этого метода и выполнение программы начинается с него. Другая особенность — всё есть объект, и наследуется от System.Object.

Создадим проект консольного приложения C#, название SharpHello.

После создания проекта файл Program.cs содержит примерно следующий код:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SharpHello {
    class Program {
        static void Main(string[] args) {
        }
    }
}
```

Вместо директивы препроцессора #include в C# используется using и далее используемые пространства имен. Основным пространством имен является System. В нем определено много других пространств, которые подключаются по мере необходимости. Visual Studio автоматически включает некоторые пространства, это не значит, что все они потребуются.

Части кода C# располагаются в пространствах имен, название которых совпадает с названием проекта, но может изменяться программистом.

Весь код располагается внутри метода Main. Сейчас этот метод содержит только оператор возврата return 0, если функция Main объявлена как int, или ничего не содержит, если функция Main объявлена как void.

Для ввода и вывода в консоль используется класс Console и его методы Write, WriteLine, Read и ReadLine. Методы со словом Line читают или записывают конец строки (символ newline).

Первое приложение начинается с вывода строки Hello, World!:

```
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Hello, World!");
        }
    }
```

Для проверки запускаем приложение при помощи Ctrl+F5.

Добавим в программу статический метод, полностью совпадающий с методом Main, за исключением того, что название метода arguments:

```
namespace SharpHello {
    class Program {
        static void Main(string[] args) {
            arguments(args);
        }
        static void arguments(string[] args) {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Следующий тест — вывод параметров функции Main. Система формирует строковый массив параметров, который передается функции Main.

Параметры командной строки передаются как массив, размер массива — это его свойство Length. Для перебора элементов массива можно использовать цикл foreach.

Начнем с простого, обычного цикла for:

```
        static void arguments(string[] args) {
            Console.WriteLine("Hello, World!");
            for (int i = 0; i < args.Length; i++) {
                Console.WriteLine(args[i]);
            }
        }
```

Чтобы протестировать этот фрагмент кода, нужно запустить приложение из командной строки, при помощи, например, FAR, и добавить при этом несколько параметров, например:

```
HelloWorld  first  second  456
```

Здесь в командной строке три параметра. Скомпилированное приложение находится в каталоге bin\Debug. Если все сделано правильно, то в консоль будет выведено:

```
Hello, World!
first
second
456
```

Для тестирования лучше установить параметры командной строки в свойствах проекта, раздел Debug, Command Arguments.

А теперь пробуем выполнить то же при помощи цикла foreach:

```
            foreach (string s in args) {
                Console.WriteLine(s);
            }
```

1.2. Чтение и запись в консоль

Метод WriteLine (Write) позволяет выводить и переменные. Для этого в том месте строки вывода, где требуется вывести переменную, ставится блок {*n*}, где *n* — номер переменной в списке, который следует за строкой через запятую. Будем выводить параметры командной строки с указанием их номера:

```
static void arguments(string[] args) {
    Console.WriteLine("Hello, World!");
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
    foreach (string s in args) {
        Console.WriteLine(s);
    }
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine("Parameter {0} is {1}", i, args[i]);
    }
}
```

Тестируем эту часть кода. Убеждаемся, что в консоль выводится:

```
Parameter 0 is first
Parameter 1 is second
Parameter 2 is 456
```

Описываем новый статический метод dialog:

```
static void Main(string[] args) {
    //arguments(args);
    dialog();
}
static void dialog() {
}
```

Чтение консоли выполняет метод ReadLine. Сначала в методе dialog запросим имя пользователя:

```
static void dialog() {
    Console.Write("Как вас называть: ");
}
```

Затем объявляем переменную s типа string и принимаем в нее то, что возвратит метод ReadLine. После этого выводим в консоль приветствие:

```
    string s;
    s = Console.ReadLine();
    Console.WriteLine("Привет, {0}!", s);
```

Далее запрашиваем возраст, принимаем его и выводим:

```
    Console.Write("Ваш возраст: ");
    s = Console.ReadLine();
    Console.WriteLine("Вам {0} лет (года).", s);
```

Запускаем программу, вводим запрашиваемые значения.
Убеждаемся, что значения переменных выводятся.

1.3. Массивы

1.3.1. Создание и вывод массивов

Создадим проект консольного приложения C#, название SharpArrays. Метод Main.
Сначала создадим прямоугольный массив:

```
// размер массива
const int size = 5;
int[,] matrix = new int[size, size];
// заполняем массив
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        matrix[i, j] = j + (i * size);
    }
}
// выводим массив
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        Console.Write(matrix[i, j] + "\t");
    }
    Console.WriteLine();
}
```

Затем создадим ломаный массив:

```
// ломаный массив
int[][] M = new int[size][];
// создаем массивы
for (int i = 0; i < M.Length; i++) {
    M[i] = new int[i + 1];
}
// выводим массивы
for (int i = 0; i < size; i++) {
    Console.Write("Строка {0} : {1}:\t", i, M[i].Length);
    for (int j = 0; j < M[i].Length; j++) {
        Console.Write(M[i][j] + " ");
    }
    Console.WriteLine();
}
```

1.3.2. Методы класса System.Array

Класс Array определяет много плезных методов: Clear — очищает диапазон массива, CopyTo — копирует массив в другой, Reverse — расставляет элементы в обратном порядке, Sort — сортирует массив (если определен интерфейс IComparer), и другие …

В Main создаем массив и пробуем найти элемент:

```
string[] N = new string[] { "123", "456", "189" };
for (int i = 0; i < N.Length; i++) {
    Console.WriteLine(N[i]);
}
// поиск
int index = Array.BinarySearch(N, "456");
Console.WriteLine("Найден элемент {0}", index);
```

Разворачиваем массив:

```
// развернем массив
Array.Reverse(N);
Console.WriteLine("Развернуто");
for (int i = 0; i < N.Length; i++) {
    Console.WriteLine(N[i]);
}
```

Сортируем массив:

```
// сортировка
Array.Sort(N);
Console.WriteLine("Сортировано");
for (int i = 0; i < N.Length; i++) {
    Console.WriteLine(N[i]);
}
```

Очистим два элемента:

```
// очистим 2 элемента
Array.Clear(N, 1, 2);
Console.WriteLine("Очищено");
for (int i = 0; i < N.Length; i++) {
    Console.WriteLine(N[i]);
}
Console.WriteLine("Размер {0}", N.Length);
```

1.4. Строки

1.4.1. Форматирование строк

Создадим проект консольного приложения C#, название SharpStrings.

Для вывода данных используются форматы преобразований, обозначаемые буквами. Метод Main.

Выведем в консоль несколько чисел в разных форматах:

```
static void strings() {
    Console.WriteLine("C  format: {0:C}", 99989.987);
    Console.WriteLine("D9 format: {0:D9}", 99999);
    Console.WriteLine("E  format: {0:E}", 99999.76543);
    Console.WriteLine("F  format: {0:F3}", 99999.9999);
    Console.WriteLine("N  format: {0:N}", 99999);
    Console.WriteLine("X  format: {0:X}", 99999);
    Console.WriteLine("x  format: {0:x}", 99999);
}
```

Например, буква C означает Currency (валюта), D9 — девять цифр, E, как и следовало ожидать — экспоненциальный формат, и т.п.

Форматирование выполняет также метод Format класса String с получением новой строки:

```
string s = String.Format("На счету {0:C}", 99989.987);
Console.WriteLine(s);
```

В качестве параметров строки вывода могут использоваться массивы, тогда индексы в блоках {} соответствуют индексам элемента массива.

Сначала нужно создать массив типа object:

```
object[] stuff = { "Hi", 2.9, 1, "there", "83" };
Console.WriteLine("stuff: {0}, {1}, {2}, {3}, {4}", stuff);
```

1.4.2. Методы класса System.String

Строки в C# являются встроенным типом данных, и они имеют кодировку Unicode. Строки происходят от System.String, и операции со строками — это методы и свойства: Length — возвращает длину строки, Concat — объединяет две строки в одну, CompareTo — сравнивает одну строку с другой, Copy — возвращает копию строки, и другие …

Строки — ссылочный тип данных, однако при их сравнении при помощи операций "==" и "!=" сравниваются значения строк, а не адреса областей памяти. Операция + перегружена так, что она автоматически вызывает статический метод Concat (конкатенация).

Описываем две строки и сравниваем их:

```
System.String strObj = "test;";
string s1 = "TEST";
// сравниваем значения строк
if (s1 == strObj) {
    Console.WriteLine("Одно и то же");
} else {
    Console.WriteLine("Разные строки");
}
```

Убеждаемся, что строки разные.

Конкатенация выполняется просто:

```
// конкатенация
string newstr = strObj + s;
Console.WriteLine("strObj + s = {0}", newstr);
```

Для доступа к элементам строки используется индексатор, как в C:

```
// индексатор для доступа к отдельному символу строки
for (int i = 0; i < s.Length; i++) {
    Console.WriteLine("Char {0} is {1}", i, s[i]);
}
```

1.4.3. Управляющие последовательности

Управляющие последовательности практически такие же, как в C:

```
// управляющие последовательности
string a, b, c;
a = "Применение \"кавычек\"";
Console.WriteLine(a);
b = "Применение \"\\\\\" и \"\\n\" - C:\\app\n.file";
Console.WriteLine(b);
// @ отменяет действие управляющих последовательностей
c = @"Последовательности отменены - C:\app\n.file";
Console.WriteLine(c);
```

В C#, как и в Java, значение строки нельзя изменить. Методы класса System.String возвращают копии строк и с этим ничего нельзя сделать.

```
// строки не изменяются
System.String fixeds = "string-String";
Console.WriteLine(fixeds);
// метод ToUpper возвращает копию строки
string uppers = fixeds.ToUpper();
Console.WriteLine(uppers);
Console.WriteLine(fixeds);
```

1.4.4. Построитель строк

Для работы со строками без создания копий нужен StringBuilder:

```
// StringBuilder работает с одной строкой
StringBuilder buff = new StringBuilder("буфер строки");
buff.Append(", который стал длиннее");
Console.WriteLine(buff);
```

После получения нужной строки используем метод ToString чтобы перевести содержимое объекта StringBuilder в обычный тип данных string:

```
// получим string из StringBuilder
string e = buff.ToString().ToUpper();
Console.WriteLine(e);
Console.WriteLine(buff);
```

1.5. Перечисления

Создадим проект консольного приложения C#.

Название проекта — SharpEnums.

Базовый тип System.Enum имеет методы и свойства, которые могут облегчить работу с перечислениями. Определим следующее перечисление:

```
namespace SharpHello {
    enum emp {
        manager = 1,
        grunt = 10,
        contractor = 100,
        VP = 99
    }
    class Program {
```

В Main выводим тип данных перечисления:

```
// тип данных перечисления
Console.WriteLine(Enum.GetUnderlyingType(typeof(emp)));
```

При помощи метода Enum.Format можно получить информацию о переменной перечисления:

```
// информация о переменной перечисления
Console.WriteLine("bill - {0}",
    Enum.Format(typeof(emp), bill, "G"));
Console.WriteLine("bill - {0}",
    Enum.Format(typeof(emp), bill, "X"));
Console.WriteLine("bill - {0}",
    Enum.Format(typeof(emp), bill, "d"));
```

Флаг форматирования G означает «вывести как строку».

При помощи метода GetValues можно получить массив элементов перечисления:

```
// информация об элементах перечисления
Array a = Enum.GetValues(typeof(emp));
Console.WriteLine("элементов {0}.", a.Length);
int i = 0;
foreach (emp e in a) {
    Console.WriteLine("Название элемента {0} - {1}", i++,
        Enum.Format(typeof(emp), e, "G"));
}
```

Свойство IsDefined:

```
// определяет наличие элемента перечисления
if (Enum.IsDefined(typeof(emp), "VP1")) {
    Console.WriteLine("Существует.");
} else {
    Console.WriteLine("Нет такого слова в перечислении.");
}
```

1.6. Структурные и ссылочные типы

Создадим проект консольного приложения C#, название SharpSAC.

Типы C# делятся на структурные типы и ссылочные типы.

К структурным типам относятся числовые типы, перечисления и структуры. Память для них выделяется из стека метода. При копировании структурного типа создается побитовая копия.

К ссылочным типам относятся классы и интерфейсы (а также строки). Память для них выделяется из кучи. При копировании ссылочного типа создается еще одна ссылка на тот же объект.

Опишем следующие структуру и класс:

```
struct sfoo {
    public int x;
}
class cfoo {
    public int x = 100;
}
```

Первое отличие структуры от класса — в структуре нельзя инициализировать элементы данных, в классе можно (и нужно). Структуры и классы C# отличаются тем, что каждый элемент имеет свой модификатор доступа.

В методе Main проведем тесты со структурой sfoo:

```
sfoo s1;
s1.x = 100;
// копия типа
sfoo s2 = s1;
// выводим структуры s1 и s2
Console.WriteLine("s1.x={0} s2.x={1}", s1.x, s2.x);
// изменим s2
s2.x = 555;
// снова выводим структуры s1 и s2
Console.WriteLine("s1.x={0} s2.x={1}", s1.x, s2.x);
```

Запустим программу. Убедимся, что объект s2 изменился.

Такие же тесты нужно выполнить с классом cfoo, называя переменные c1 и c2:

```
// классы
cfoo c1 = new cfoo();
// копия типа
cfoo c2 = c1;
// выводим структуры c1 и c2
Console.WriteLine("c1.x={0} c2.x={1}", c1.x, c2.x);
// изменим c2
c2.x = 555;
// снова выводим структуры c1 и c2
Console.WriteLine("c1.x={0} c2.x={1}", c1.x, c2.x);
```

Второе отличие структуры от класса — структуру не обязательно создавать при помощи new, а представитель класса создается только при помощи new.

1.7. Все есть объект

Создадим проект консольного приложения C#.

Название проекта — SharpObjects.

В .NET все есть объект, производный от System.Object, и все объекты имеют методы ToString, GetHashCode, GetType, Equals.

В Main выводим свойства разных объектов:

```
static void isobject() {
    int i = 128;
    Console.WriteLine("{0}", i.ToString());
    Console.WriteLine("{0}", 256.ToString());
    Console.WriteLine("{0}", 256.GetTypeCode());
    Console.WriteLine("{0}", i.GetType());
    Console.WriteLine("{0}", i.GetTypeCode());
    Console.WriteLine("{0}", int.MaxValue);
    Console.WriteLine("{0}", int.MinValue);
    Program p1 = new Program();
    // Информация об объекте
    Console.WriteLine("ToString: {0}", p1.ToString());
    Console.WriteLine("HashCode: {0}", p1.GetHashCode());
    Console.WriteLine("Type: {0}", p1.GetType());
    // создаем еще одну ссылку на p1
    Program p2 = p1;
    // оба экземпляра указывают на одну область памяти?
    if (p1.Equals(p2)) {
        Console.WriteLine("Тот же экземпляр!");
    }
}
```

1.7.1. Замещение методов SystemObject

Создадим проект консольного приложения C#.

Название проекта — SharpObjectOverride.

Описываем класс Person:

```
namespace SharpObjectOverride {
    class Person {
        public string name = "";
        public string ssn = "";  // номер социального страхования
        public byte age = 1;
        public Person() {}
        public Person(string n, string s, byte a) {
            name = n;
            ssn = s;
            age = a;
        }
    }
    struct sfoo {
```

Замещаем в классе Person методы Object:

```
        public override string ToString() {
            StringBuilder sb = new StringBuilder();
            sb.Append("[Name = " + this.Name);
            sb.Append(" SSN = " + this.SSN);
            sb.Append(" Age = " + this.age + "]");
            return sb.ToString();
        }
        public override bool Equals(object o) {
            Person temp = (Person)o;
            bool eq = temp.Name == this.Name;
            eq &= temp.SSN == this.SSN;
            eq &= temp.age == this.age;
            if (eq) {
                return true;
            } else {
                return false;
            }
        }
        public override int GetHashCode() {
            return SSN.GetHashCode();
        }
```

Метод Main:

```
            Person p1 = new Person("Fred", "222-22-2222", 98);
            Person p2 = new Person("Fred", "222-22-2222", 98);
            if (p1.Equals(p2) && p1.GetHashCode() == p2.GetHashCode()){
                Console.WriteLine("P1 и P2 одинаковы");
            } else {
                Console.WriteLine("P1 и P2 разные");
            }
            // меняем состояние p2
            p2.age = 2;
            // сравниваем заново
            if (p1.Equals(p2) && p1.GetHashCode() == p2.GetHashCode()){
                Console.WriteLine("P1 и P2 одинаковы");
            } else {
                Console.WriteLine("P1 и P2 разные");
            }
            // используем замещенный метод ToString:
            Console.WriteLine(p1.ToString());
            // WriteLine вызывает метод ToString автоматически
            Console.WriteLine(p2);
```

Кроме виртуальных методов, System.Object имеет два статических метода Equals и ReferenceEquals:

```
// статические члены System.Object
Person p3 = new Person("Sally", "333", 4);
Person p4 = new Person("Sally", "333", 4);
// одинаково ли внутреннее состояние p3 и p4? Да!
Console.WriteLine("P3 и P4 одинаковы: {0}",
    object.Equals(p3, p4));
// представляют ли они один и тот же объект в памяти, Нет!
Console.WriteLine("P3 и P4 одно и то же: {0}",
    object.ReferenceEquals(p3, p4));
```

Запускаем программу, убеждаемся, что p3 и p4 одинаковы, но не одно и то же.

1.8. Случайные числа

Создадим проект консольного приложения C#.
Название проекта — SharpRandom.
Описываем класс Teenager:

```
class Teenager {
    // для генерации случайных чисел
    private Random r = new Random();
}
```

Опишем метод Complain (в классе Teenager):

```
public string Complain() {
    string[] messages = new string[5] {
        "А я должен?", "Он первый начал...", "Я устал...",
        "Ненавижу школу!", "Вы тааак заблуждаетесь!"
    };
    return messages[GetRandomNumber(5)];
}
```

Опишем метод GetRandomNumber (в классе Teenager):

```
public int GetRandomNumber(int upperLimit) {
    // Random.Next() возвращает случайное целое число
    // в диапазоне от 0 до upperLimit
    return r.Next(upperLimit);
}
```

Метод Main:

```
Teenager mike = new Teenager();
// mike, начинай жаловаться
for (int i = 0; i < 12; i++) {
    Console.WriteLine(mike.Complain());
}
```

1.9. Статические методы класса Environment

Создадим проект консольного приложения C#.
Название проекта — SharpEnviron.
Метод Main:

```
// операционная система
Console.WriteLine("OS: {0}", Environment.OSVersion);
// текущий каталог
Console.WriteLine("Dir: {0}",
    Environment.CurrentDirectory);
// список логических дисков
string[] drives = Environment.GetLogicalDrives();
for (int i = 0; i < drives.Length; i++) {
    Console.WriteLine("Disk {0} : {1}", i, drives[i]);
}
// версия платформы .NET
Console.WriteLine(".NET: {0}", Environment.Version);
```

1.10. Контрольные вопросы и упражнения

1. Опишите сигнатуру метода Main.
2. Как получить доступ к аргументам командной строки?
3. Опишите вывод и форматированный вывод переменных.
4. Перечислите структурные и ссылочные типы.
5. Чем структурные типы отличаются от ссылочных?
6. Назовите общие методы всех объектов.

2. Работа CS-102. Классы в C#

Цели:
- классы C#.
Задачи:
- создание структур и классов;
- наследование и полиморфизм;
- пространства имен.

Опорные документы:

2.1. Определение структуры

Создадим проект консольного приложения C#.
Название проекта — SharpStruct.
Структуры в C# являются структурным типом данных, но при этом могут обладать методами. Объявим перед классом программы перечисление типов работников:

```
enum emp_type : byte {
    manager = 10, grunt = 1, contractor = 100, VP = 9
}
```

Заметим, что мы указываем тип данных перечисления byte.
Ниже объявим структуру, описывающую работника:

```
struct sEmployee {
    public emp_type title;
    public string name;
    public short deptID;
}
```

Структура C# может иметь конструктор, но только с параметрами.
Описываем его в структуре sEmployee:

```
    // конструктор
    public sEmployee(emp_type et, string n, byte d) {
        title = et;
        name = n;
        deptID = d;
    }
```

Опишем метод для вывода информации о работнике (перед Main):

```
    void DisplaysEmloyeeStats(sEmployee e) {
        Console.WriteLine("{0}: отдел {1}, тип {2}",
            e.name, e.deptID,
            Enum.Format(typeof(emp_type), e.title, "G") + ".");
    }
```

В Main создаем представителя класса программы и работника bill:

```
Program t = new Program();
sEmployee bill = new sEmployee(emp_type.VP, "Bill", 10);
t.DisplaysEmloyeeStats(bill);
```

Запустим программу, убедимся, что сведения выводятся.

Для преобразования структурного типа в ссылочный тип используется упаковка. Упаковка выполняется просто: присвоением структурного типа ссылочному типу (метод Main):

```
// упаковка структурного типа в ссылочный
object bill_inbox = bill;
```

Опишем еще один метод класса программы, который распаковывает ссылочный тип в структурный тип. Распаковка также выполняется просто — приведением к структурному типу:

```
void UnboxsEmployee(object o) {
    // распаковываем ссылочный тип в структурный
    sEmployee e = (sEmployee)o;
    DisplaysEmloyeeStats(e);
}
```

В функции Main вызываем новый метод дважды:

```
// упакованный
t.UnboxsEmployee(bill_inbox);
// компилятор производит упаковку автоматически
t.UnboxsEmployee(bill);
```

2.2. Определение класса

Создадим проект консольного приложения C#, название SharpClass.

Класс в C# определяется примерно так же, как и в C++:

```
class Employee {
    // закрытые данные класса
    private string fullName;
    private short empID;
    private double currPay;
}
```

Если в классе определен конструктор с параметрами, определение конструктора по умолчанию (без параметров) является обязательным:

```
// конструктор по умолчанию
public Employee() {}
// конструктор преобразования
public Employee(string fullName, short empID, double currPay) {
    this.fullName = fullName;
    this.empID = empID;
    this.currPay = currPay;
}
```

Методы класса определяются обычным образом:

```
// метод для увеличения зарплаты
public void GiveBonus(double amount) {
    currPay += amount;
}
```

А это виртуальный метод:

```
// выводит сведения о текущем состоянии
public virtual void DisplayStats() {
    Console.WriteLine("\nИмя: {0}", fullName);
    Console.WriteLine("Зарплата: {0}", currPay);
    Console.WriteLine("Табельный номер: {0}", empID);
}
```

Код метода Main определяет двух сотрудников:

```
static void Main(string[] args) {
    Employee e = new Employee("Bill", 100, 30000.0);
    e.GiveBonus(500.0);
    e.DisplayStats();
}
```

Запускаем программу, убеждаемся, что сведения выводятся.
Второй сотрудник:

```
static void Main(string[] args) {
    Employee e = new Employee("Bill", 100, 30000.0);
    e.GiveBonus(500.0);
    e.DisplayStats();
    Employee e2;
    e2 = new Employee("Mary", 101, 36000.0);
    e2.GiveBonus(1000.0);
    e2.DisplayStats();
}
```

2.3. Внутренние классы

Нововведение C#, связанное с организацией .NET — внутренние классы и элементы. Внутренние классы можно использовать только внутри текущей сборки.

Перед классом Employee опишем один такой внутренний класс:

```
internal class EmployeeHelper {
    private string helpstring;
    public EmployeeHelper() { }
    public EmployeeHelper(string hs) {
        helpstring = hs;
    }
}
```

Внутренними могут быть также структуры, перечисления, методы и свойства. Опишем в элементах данных класса Employee ссылку на класс EmployeeHelper:

```
class Employee {
    // закрытые данные класса
    private EmployeeHelper helper;
    private string fullName;
    . . .
}
```

В этом нет никакого смысла, просто так…

2.4. Свойства в классах

Новым является также возможность определять не только методы классов, но свойства. Свойство задается процедурами get (чтение) и set (запись нового значения). Их еще называют аксессорами (от access) или же геттерами и сеттерами. Новое значение свойства при этом указывается специальным словом value.

Опишем свойство Name для класса Employee:

```
// свойство Имя
public string Name {
    get {
        return fullName;
    }
    set {
        fullName = value;
    }
}
```

Используем свойство Name в функции Main:

```
static void Main(string[] args) {
    Employee e = new Employee("Bill", 100, 30000.0);
    e.GiveBonus(500.0);
    e.DisplayStats();
    Employee e2;
    e2 = new Employee("Mary", 101, 36000.0);
    e2.GiveBonus(1000.0);
    e2.DisplayStats();
    // используем свойство класса
    e2.Name = "Ann";
    e2.DisplayStats();
    . . .
}
```

Обратим внимание на отсутствие скобок () после имени свойства.

Самостоятельно опишем свойство Salary (зарплата).

Свойства могут быть только для чтения и только для записи.

Пример свойства только для чтения:

```
// свойство табельный номер только для чтения
public short ID {
    get {
        return empID;
    }
}
```

(Заметим, что в современной версии C# свойства, описанные простым способом (не имеющие алгоритмов), можно указывать только именами аксессоров, а код можно не писать.)

В функции Main выводим значение свойства для e2:

```
Console.WriteLine("Табельный номер e2: {0}", e2.ID);
```

Запустим программу.

2.5. Статический конструктор

Еще одно нововведение C# — статические конструкторы. Их единственное назначение — задавать значения статическим элементам данных.

Определим в классе Employee статический элемент данных, задающий название компании:

```
class Employee {
    // название организации
    private static string company_name;
    // название организации
    private static string company_name;
    . . .
}
```

Опишем статический конструктор в классе Employee:

```
    // статический конструктор
    static Employee() {
        company_name = "ОТИ МИФИ";
    }
```

Опишем метод, возвращающий название организации:

```
    // метод возвращает название организации
    public string GetCompanyName() {
        return company_name;
    }
```

В методе Main выведем в консоль название организации:

```
        Console.WriteLine("Метод: {0}", e.GetCompanyName());
```

Мы можем также определить статическое свойство в классе Employee:

```
    // мвойство возвращает название организации
    public static string Company {
        get {
            return company_name;
        }
    }
```

В конце метода Main используем это свойство:

```
        Console.WriteLine("Свойство: {0}", Employee.Company);
```

2.6. Наследование и полиморфизм

Рассмотрим примеры наследования.

Объявим новый класс, наследующий Employee:

```
class Manager : Employee {
    // менеджер должен знать количество опционов
    private ulong numberOfOpoints;
}
```

Опишем в классе Manager свойство OptionsNumber:

```
    // свойство количество опционов
    public ulong OptionsNumber {
        get {
            return numberOfOpoints;
        }
        set {
            numberOfOpoints = value;
        }
    }
```

В Main создадим представителя класса Manager:

```
        Manager m = new Manager();
```

Толку в этом пока нет — нельзя задать значения элементов данных. Изменим тип доступа к элементам данных класса Employee:

```
class Employee {
    // название организации
    protected static string company_name;
    // закрытые данные класса
    protected EmployeeHelper helper;
    protected string fullName;
    protected short empID;
    protected double currPay;
    . . .
}
```

В классе Manager опишем конструкторы:

```
    // конструктор по умолчанию
    public Manager() {
    }
    // конструктор преобразования
    public Manager(short empID) {
        this.empID = empID;
    }
```

Теперь вернемся в функцию Main и изменим создание представителя менеджеров:

```
        Manager m = new Manager(102);
```

Значения других элементов класса зададим через свойства, и выведем значения при помощи DisplayStats:

```
        m.Name = "John";
        m.Salary = 60000.0;
        // информация о менеджере
        ((Employee)m).DisplayStats();
```

Определим еще один конструктор класса Manager, используя конструктор базового класса base:

```
    // еще один конструктор класса Manager
    public Manager(string name, short id, double pay, ulong p)
        : base(name, id, pay) {
        numberOfOpoints = p;
    }
```

Создадим нового представителя класса Manager с использованием нового конструктора, конец функции Main:

```
Manager m2 = new Manager("Sam", 103, 40000.0, 20);
((Employee)m2).DisplayStats();
```

Переопределим виртуальный метод DisplayStats для класса Manager, используя определенный ранее метод базового класса (base):

```
// выводит сведения о текущем состоянии
public override void DisplayStats() {
    base.DisplayStats();
    Console.WriteLine("Опционов: {0}", numberOfOpoints);
}
```

Теперь вместо строк вида

```
((Employee)m).DisplayStats();
```

можно использовать строки вида

```
m.DisplayStats();
```

Определим еще один производный класс для продавца.
Продавец должен знать объем продаж:

```
class Salesman : Employee {
    private ulong numberOfSales;
}
```

Определим конструкторы, аналогичные конструкторам Manager:

```
// конструкторы
public Salesman() { }
public Salesman(string name, short id, double pay, ulong p)
    : base(name, id, pay) {
    numberOfSales = p;
}
```

Опишем виртуальный метод DisplayStats для продавца:

```
// выводит сведения о текущем состоянии
public override void DisplayStats() {
    base.DisplayStats();
    Console.WriteLine("Продаж: {0}", numberOfSales);
}
```

В функции Main создаем нового продавца и выводим его данные:

```
Salesman s = new Salesman("Stan", 201, 20000, 10);
s.DisplayStats();
```

Создадим еще один класс, производный от класса продавца — продавец на неполный рабочий день (part-time salesman). Мы хотим также, чтобы от нового класса нельзя было далее продолжать наследование. В этом случае класс определяется со словом sealed:

```
    // класс, заканчивающий цепочку наследования
    sealed class POINTSalesman : Salesman {
    }
```

Данный класс не может выступать в качестве базового.

Определим конструкторы нового класса:

```
        // конструкторы
        public POINTSalesman() { }
        public POINTSalesman(string name, short id, double pay, ulong
p)
            : base(name, id, pay, p) {
        }
```

В функции Main создадим представителя и выведем его данные:

```
            POINTSalesman points = new POINTSalesman("WhoAmI", 301,
10000, 8);
            points.DisplayStats();
```

2.7. Пространства имен

Создадим проект консольного приложения C#.

Название проекта — SharpNS.

Пространство имен — это логическая структура для организации имен. Они позволяют сгруппировать определяемые типы данных.

Определим три пространства имен:

```
namespace baseshapes {
}
namespace shapes {
}
namespace shapes {
}
```

Обратим внимание, что два пространства имен названы одинаково.

В первом пространстве имен располагаем описание класса figure:

```
namespace baseshapes {
    class figure {
    }
}
```

Во втором пространстве имен определим класс circle:

```
namespace shapes {
    using baseshapes;
    class circle : figure {
    }
}
```

В третьем пространстве имен определим класс square.

Опишем в классе figure абстрактный метод:

```
    class figure {
        virtual public void draw() {}
    }
```

В классах circle и square опишем реализацию метода draw, используя модификатор override. Пусть методы выводят в консоль названия классов.

В функции Main можно создать представителя класса circle:

```
class Program {
    static void Main(string[] args) {
        shapes.circle c = new shapes.circle();
        c.draw();
    }
}
```

При помощи using то же самое сделать будет проще. Указываем использование пространства имен в начале модуля:

```
using shapes;
```

В методе Main можно не использовать пространство имен shapes:

```
class Program {
    static void Main(string[] args) {
        shapes.circle c = new shapes.circle();
        c.draw();
        square s = new square();
        s.draw();
    }
}
```

2.8. Контрольные вопросы и упражнения

1. Чем структура отличается от класса?
2. Что называется статическим конструктором?
3. Для чего нужен статический конструктор?
4. Как определяются методы класса?
5. Как определяются свойства класса?
6. Чем свойство отличается от метода?
7. Как определяется наследование классов?
8. Как переопределяется виртуальный метод класса?

3. Работа CS-103. Индексаторы

Цели:
- изучение индексаторов C#.
Задачи:
- описание простого индексатора;
- описание составного индексатора.

Опорные документы:

3.1. Простой индексатор

Создадим проект консольного приложения C#.
Название проекта — SharpIndexer.
Индексатор — это пара методов, похожих на пару методов свойства, отличающаяся от последних наличием параметров. С помощью индексаторов объекты класса можно индексировать, как элементы массива.
Опишем класс для работы с множеством элементов типа string:

```
class Stringer {
    private string[] ss;
    public Stringer(int n) {
        ss = new string[n];
    }
}
```

Для работы с отдельными строками класса можно создать два метода, например:

```
    public string GetItem(int m) {
        return ss[m];
    }
    public void SetItem(int m, string newVal) {
        ss[m] = newVal;
    }
```

В методе Main можно создать представителя класса Stringer, задать какие-нибудь значения, и вывести их:

```
    static void Main(string[] args) {
        Stringer s = new Stringer(2);
        s.SetItem(0, "Ann");
        s.SetItem(1, "Mary");
        Console.WriteLine(s.GetItem(0));
        Console.WriteLine(s.GetItem(1));
    }
```

В принципе, задача решена.
Однако можно заметить, что описание методов класса похоже на методы, которые в C++ используются для описания свойств. Отсюда, наверное, и родилась идея индексаторов. Можно сказать, что индексаторы — это свойства с параметрами.

Однако их определение в классе задает определенное его поведение.

Добавим в класс Stringer индексатор:

```
// индексатор
public string this[int m] {
    get {
        return ss[m];
    }
    set {
        ss[m] = value;
    }
}
```

Теперь в методе Main мы можем использовать представителя класса Stringer так, как будто это элемент массива, например:

```
s[0] = "John";
Console.WriteLine(s[0]);
```

Можно также создавать интерфейсы, которые содержат индексаторы.

Определим такой интерфейс:

```
interface StringIndexer {
    string this[int m] { get; set;}
}
```

Наследуем интерфейс StringIndexer в классе Stringer:

```
class Stringer : StringIndexer {
```

Больше ничего не нужно изменять.

Наследуемый индексатор уже реализован в классе Stringer.

Индексаторы могут иметь произвольное количество параметров произвольного типа, не обязательно целочисленного.

Логика работы индексатора может быть произвольной.

3.2. Составной индексатор

Создадим проект консольного приложения C#.

Название проекта — SharpMatrix.

Назовем индексатор составным, если в нем более одного индекса.

В качестве упражнения вам предлагается описать класс матрицы произвольной размерности Matrix с индексатором из двух индексов (первый по горизонтали, второй по вертикали).

Методы индексатора должны проверять допустимость индексов.

После определения класса Matrix создайте интерфейс для класса матрицы IMatrix, и опишите наследование этого интерфейса.

3.3. Контрольные вопросы и упражнения

1. Опишите назначение индексатора.
2. Опишите синтаксис индексатора.

4. Работа CS-104. Исключения

Цели:
- обработка исключения.
Задачи:
- системные исключения;
- пользовательские исключения;

Опорные документы:

4.1. Генерация и перехват исключений

Создадим проект консольного приложения C#.
Название проекта — SharpException.
Исключения в C# — объекты, производные от System.Exception.
Наиболее важные свойства этого типа:
HelpLink — URL файла справки;
Message — описание ошибки;
Source — объект, сгенерировавший ошибку;
StackTrace — последовательность вызовов методов, которые привели к возникновению исключения.
Опишем класс Car:

```
class Car {
    private int speed;
    private int maxSpeed;
    private string petName;
    bool dead;
}
```

Конструктор по умолчанию:

```
    public Car() {
        maxSpeed = 100;
        dead = false;
    }
```

Конструктор с параметрами:

```
    public Car(string name, int max, int curr) {
        maxSpeed = max;
        speed = curr;
        petName = name;
        dead = false;
    }
```

Метод, предназначенный для разгона:

```
    public void SpeedUp(int delta) {
    }
```

Описываем в нем приращение скорости:

```
public void SpeedUp(int delta) {
    if (dead) {
        Console.WriteLine(petName + " is dead...");
    } else {
        speed += delta;
        if (speed < maxSpeed) {
            Console.WriteLine("\tCurrent Speed = " + speed);
        } else {
            Console.WriteLine(petName + " has overheated...");
            dead = true;
        }
    }
}
```

Создаем новый представитель класса Car в Main:

```
Car c1 = new Car("boom", 100, 0);
for (int i = 0; i < 10; i++) {
    c1.SpeedUp(20);
}
```

Теперь будем генерировать исключение при превышении максимальной скорости:

```
public void SpeedUp(int delta) {
    if (dead) {
        throw new Exception("This car is already dead");
        //Console.WriteLine(petName + " is dead...");
    } else {
```

В методе Main перехватываем исключение при помощи try и catch:

```
try {
    for (int i = 0; i < 10; i++) {
        c1.SpeedUp(20);
    }
}
catch (Exception e) {
    Console.WriteLine(e.Message);
    Console.WriteLine(e.StackTrace);
}
```

Запускаем программу, наблюдаем выброс исключения и трассировку методов, приведших к исключению.

4.2. Пользовательские исключения

Можно использовать класс Exception, для того, чтобы сгенерировать исключение. Однако часто удобнее создать собственный класс исключения с необходимыми методами.

Закомментируем строку:

```
catch (Exception e) {
    Console.WriteLine(e.Message);
    //Console.WriteLine(e.StackTrace);
}
```

Опишем новый класс исключения:

```
class CarIsDeadException : Exception {
    // с его помощью мы получим имя сдохшей машины
    private string carName;
}
```

Конструкторы класса CarIsDeadException:

```
    public CarIsDeadException() { }
    public CarIsDeadException(string carName) {
        this.carName = carName;
    }
```

Замещаем свойство Message:

```
    public override string Message {
        get {
            string msg = base.Message;
            if (carName != null) {
                msg += "\n" + carName + " has been destroyed";
            }
            return msg;
        }
    }
```

Генерируем новое исключение в методе SpeedUp:

```
        if (dead) {
            throw new CarIsDeadException(this.petName);
            //throw new Exception("This car is already dead");
```

Запускаем программу. Исследуем вывод в консоль.

Можно придумать какой угодно класс для генерации пользовательских исключений.

Однако чем проще, тем лучше, например, так:

```
class CarIsDeadException2 : Exception {
    public CarIsDeadException2() {
    }
    public CarIsDeadException2(string message)
        : base(message) {
    }
}
```

Выбрасываем новое исключение:

```
        if (dead) {
            throw new CarIsDeadException2(this.petName
                + " is dead now");
            //throw new CarIsDeadException(this.petName);
```

Запускаем программу. Исследуем вывод в консоль.

4.3. Делегирование методов

Создадим новый класс — автомобильное радио:

```
class Radio {
    public Radio() { }
}
```

Добавим в класс Radio метод для включения/выключения:

```
    public void TurnOn(bool on) {
        if (on) {
            Console.WriteLine("Radio is now on");
        } else {
            Console.WriteLine("Radio is now quiet");
        }
    }
```

Реализуем модель между классами включение-делегирование:

```
class Car {
    private Radio theMusicBox = new Radio();
    private int speed;
```

Делегируем метод класса Radio классу Car (метод класса Car):

```
    public void Tune(bool state) {
        theMusicBox.TurnOn(state);
    }
```

Создав представителя Car, включим радио:

```
        Car c1 = new Car("boom", 100, 0);
        c1.Tune(true);
```

Перед завершением метода Main выключим радио (после блока catch):

```
        c1.Tune(false);
```

Интересно, когда радио уничтожается?
Опишем деструктор класса Radio:

```
    ~Radio() {
        Console.WriteLine("Radio is now destroyed");
    }
```

Запускаем программу. Наблюдаем момент уничтожения радио.
Опишем еще одно пользовательское исключение:

```
class CarInvalidSpeedUp : Exception {
    public CarInvalidSpeedUp() {
    }
    public CarInvalidSpeedUp(string message)
        : base(message) {
    }
}
```

Выбрасываем новое исключение в методе SpeedUp:

```
        } else {
            if (delta < 0) {
                // выбрасываем системное исключение
                throw new ArgumentOutOfRangeException(""
                    + "Must be greater than zero");
            } else if (delta > 50) {
                throw new CarInvalidSpeedUp(""
                    + "Invalid acceleration for " + petName);
            }
            speed += delta;
            . . .
```

Запускаем программу. Тестируем код, используя разные параметры delta в цикле, например, 60 и –20.

Будем отлавливать разные исключения разными блоками catch:

```
} catch (ArgumentOutOfRangeException e) {
    Console.WriteLine("ArgumentOutOfRangeException "
        + e.Message);
} catch (CarInvalidSpeedUp e) {
    Console.WriteLine("CarInvalidSpeedUp " + e.Message);
} catch (CarIsDeadException2 e) {
    Console.WriteLine("CarIsDeadException2 " + e.Message);
}
c1.Tune(false);
```

Пробуем отлавливать разные исключения, используя разные значения параметра delta.

Наконец, опишем блок finally (после последнего блока catch):

```
} finally {
    Console.WriteLine("Finally in");
    c1.Tune(false);
    Console.WriteLine("Finally out");
}
```

4.4. Контрольные вопросы и упражнения

1. Что называется исключением?
2. Назовите элементы класса Exception.
3. Опишите структуру обработки исключения try-catch-finally. Для чего предназначен каждый блок этой структуры?

5. Работа CS-105. Абстрактные классы и интерфейсы

Цели:
- абстрактные классы и интерфейсы.
Задачи:
- абстрактные классы;
- абстрактные методы;
- наследование абстрактных классов;
- наследование интерфейсов.

Опорные документы:

5.1. Базовый класс

Создадим проект консольного приложения C#.

Название проекта — SharpShapes.

Наша первая цель — воспроизвести иерархию классов геометрических фигур Shape — Circle, Square.

Класс Shape. Опишем следующие элементы класса:
- защищенное поле name типа string, значение по умолчанию "figure",
- конструктор по умолчанию,
- конструктор с параметром типа string, задает поле name,
- не виртуальный метод Draw, выводит в консоль "Shape имя"
- свойство Name.

После определения класса создадим в методе Main представителя, вызовем метод Draw. Убедимся, что в консоль выводится название класса и имя объекта.

Замечание по формированию конструктора. В C# в конструкторе есть список инициализации, как и в C++, однако в нем можно вызывать только конструкторы. Конструктор базового класса вызывается при помощи ключевого слова base, другой конструктор этого же класса вызывается при помощи ключевого слова this.

Вы можете попробовать создать конструктор с параметром, задающим поле name, следующим образом:

```
class Shape {
    public Shape(string n) : name(n) {
    }
}
```

При этом компилятор будет сообщать об ошибке.

Так нельзя.

С другой стороны, вы можете определить конструктор с двумя параметрами. Для этого добавьте в класс закрытое поле ID целого типа, значение по умолчанию нулевое.

Теперь можно задать два следующих конструктора:

```
public Shape(string name) {
    this.name = name;
}
public Shape(string name, int id) : this(name) {
    ID = id;
}
```

Здесь this(name) вызывает конструктор с одним параметром.

5.2. Абстрактный класс

Сделаем класс Shape абстрактным.

Для этого к описанию класса добавим модификатор abstract:

```
abstract class Shape {
```

Убедимся, что создать представителя класса теперь нельзя, поэтому удалим из Main создание представителя. Абстрактный класс может служить только в качестве базового класса. Поэтому далее описываем два производных класса Circle и Square. Методы Draw этих классов выводят в консоль "Circle имя" и "Square имя" соответственно.

Заметим, что методы Draw не могут иметь модификатор override, этот модификатор допустим, только если соответствующий метод базового класса имеет модификатор virtual, abstract или override.

В Main создаем представителей классов и вызываем их методы Draw.

Убеждаемся, что методы выводят правильное название класса:

```
Circle circle
Square square
```

Теперь попробуем следующий код:

```
Shape fc = new Circle("circle");
fc.Draw();
Shape fs = new Square("square");
fs.Draw();
```

Убедимся, что в консоль выводятся строки:

```
Circle circle
Square square
Shape circle
Shape square
```

Этого и следовало ожидать. Теперь включим полиморфизм.

В метод Draw базового класса добавим модификатор virtual, в методы Draw производных классов добавим модификатор override.

Убедимся, что выводятся строки:

```
Circle circle
Square square
Circle circle
Square square
```

Таким образом, иерархия классов фигур реализована.

5.3. Абстрактный метод

В абстрактном классе можно определить абстрактный метод.

Пробуем приписать модификатор abstract к методу Draw базового класса Shape. Компилятор сообщит об ошибке.

Во-первых, метод не должен иметь реализации:

```
public abstract virtual void Draw();
```

Компилятор все равно не допускает этот код.

Во-вторых, абстрактный метод виртуальный по определению, поэтому нужно удалить модификатор virtual, — модификаторы abstract и virtual взаимоисключающие, можно использовать либо тот, либо другой.

После удаления модификатор virtual программа снова заработает.

Интересно также узнать, можно ли определить абстрактный метод в не абстрактном базовом классе. Пробуем удалить модификатор abstract у класса Shape:

```
class Shape {
```

Убедимся, что компилятор сопротивляется.

Нельзя определять абстрактный метод в не абстрактном классе.

5.4. Интерфейсы

Класс C# может наследовать только один базовый класс, множественное наследование классов исключено. Однако разрешено множественное наследование интерфейсов.

Интерфейс определяется иначе, чем абстрактный класс.

Отличие проявляется в том, что методы интерфейса являются абстрактными (и виртуальными) по определению, поэтому никакие модификаторы в описании методов недопустимы, даже public. Кроме методов, интерфейс может содержать только свойства, события и индексаторы.

Первый интерфейс, который мы определим, это IDrawable.

Этот интерфейс определяет только метод Draw:

```
interface IDrawable {
    void Draw();
}
```

Описание интерфейса, как видим, максимально упрощено.

Теперь описываем наследование интерфейса классом Shape:

```
abstract class Shape : IDrawable {
```

Удивительно, но компилятор не сопротивляется нашим действиям.

Попробуйте закомментировать метод Draw класса Shape. Для чистоты эксперимента нужно также закомментировать описание производных классов и код метода Main.

Теперь компилятор сопротивляется.

Если в интерфейсе есть метод, в производном абстрактном классе он должен быть абстрактным или иметь реализацию.

Абстрактным метод Draw класса Shape является сейчас.

Пусть производные классы описаны.

Попробуем приписать реализацию методу Draw класса Shape:

```
public virtual void Draw() {
    Console.WriteLine("Shape {0}", name);
}
```

Компилятор допускает такую форму метода.

Таким образом, абстрактный класс, наследующий интерфейс, может определять либо абстрактный, либо виртуальный метод. Модификатор override в этом случае в классе Shape недопустим.

Вернем назад абстрактный метод Draw класса Shape:

```
public abstract void Draw();
```

Опишем интерфейс IPointy (угловатый):

```
interface IPointy {
    // абстрактный элемент интерфейса
    int Points();
}
```

Описываем наследование этого интерфейса классом Square, потому что квадрат имеет четыре угла:

```
class Square : Shape, IPointy {
```

Теперь мы обязаны описать метод Points в классе Square, метод возвращает значение 4. Реализация должна отличаться от метода интерфейса обязательным наличием модификатора public.

Экспериментируем в Main:

```
Console.WriteLine("points = {0}", s.Points());
Console.WriteLine("points = {0}", fs.Points());
```

Выясняем, что вторая строка недопустима. Объект s имеет тип Square, объект fs имеет тип Shape.

5.5. Определение наличия интерфейса

Как узнать, что некоторый объект реализует некоторый интерфейс?

Есть несколько способов.

Способ 1. Пробуем привести объект к типу интерфейса. Если приведение невозможно, возникает исключение InvalidCastException, поэтому мы должны использовать блоки try и catch:

```
IPointy ip;
try {
} catch (InvalidCastException) {
    Console.WriteLine("No IPointy");
}
```

Это полигон для испытания.

В блок try вписываем сначала приведение объекта s к типу IPointy и пробуем присвоить это переменной ip типа интерфейса:

```
ip = (IPointy)s;
Console.WriteLine("points = {0}", ip.Points());
```

Убедимся, что этот фокус удается. Теперь попробуем сделать то же самое для объекта fs, затем для объекта с класса Circle. Результат тестирования записываем в отчет.

Способ 2. Используем ключевое слово as. Полигон для этого испытания выглядит следующим образом:

```
ip = s as IPointy;
if (ip == null) {
    Console.WriteLine("No as IPointy");
} else {
    Console.WriteLine("points = {0}", ip.Points());
}
```

Тестирование заключается в использовании вместо переменной s в приведенном фрагменте кода всех других переменных.

Способ 3. Используем ключевое слово is в условии. Пример кода для объекта s имеет следующий вид:

```
if (s is IPointy) {
    Console.WriteLine("points = {0}", ((IPointy)s).Points());
} else {
    Console.WriteLine("No is IPointy");
}
```

Обратим внимание, что объект в некоторых случаях требует приведения к типу IPointy. В некоторых случаях приведение не обязательно.

Опишем еще один интерфейс:

```
public interface IDraw3D {
    void Draw();
}
```

Пусть класс Square наследует два интерфейса:

```
class Square : Shape, IPointy, IDraw3D
```

Поскольку Square переопределяет метод Draw, следующий код приводит к двусмысленности:

```
if (s is IDraw3D) {
    ((IDraw3D)s).Draw();
} else {
    Console.WriteLine("No IDraw3D");
}
```

Здесь используется имеющийся метод Draw класса Square.

В этом случае нужно переопределить метод интерфейса IDraw3D с использованием квалификатора следующим образом:

```
    // IDraw3D
    void IDraw3D.Draw() {
        Console.WriteLine("Square3D {0}", name);
    }
```

Теперь все выводится правильно.

5.6. Интерфейсы как параметры методов

Указатели на интерфейсы могут быть использованы как параметры методов. Опишем метод для рисования трехмерных объектов, параметр метода — указатель на интерфейс IDraw3D:

```
class Program {
    public static void DrawShapeIn3D(IDraw3D itf3d) {
        Console.WriteLine("*** DrawShapeIn3D ***");
        itf3d.Draw();
    }
```

Теперь в Main можно вызвать эту функцию для тех объектов, которые поддерживают интерфейс IDraw3D, например:

```
        if (s is IDraw3D) {
            ((IDraw3D)s).Draw();
            DrawShapeIn3D((IDraw3D)s);
        } else {
            Console.WriteLine("No IDraw3D");
        }
```

5.7. Иерархия интерфейсов

Создадим проект консольного приложения C#.

Название проекта — SharpInterfaces.

Интерфейсы могут образовывать иерархии. Если класс наследует один из интерфейсов иерархии, то он должен переопределить все методы всех интерфейсов вверх по иерархии.

Опишем иерархию следующих интерфейсов:

```
// иерархия интерфейсов
interface IDraw {
    void Draw();
}
interface IDraw2 : IDraw {
    void DrawToPrinter();
}
interface IDraw3 : IDraw2 {
    void DrawToMetafile();
}
```

Опишем класс, который наследует все интерфейсы иерархии:

```
class SuperImage : IDraw3 {
}
```

Класс SuperImage должен определять методы всех интерфейсов.

Описываем все методы.

Пусть методы выводят следующие строки:

IDraw.Draw

IDraw2.DrawToPrinter

IDraw3.DrawToMetafile

В методе Main создаем представителя SuperImage, и описываем полигон для испытаний:

```
SuperImage si = new SuperImage();
try {
}
catch (InvalidCastException) {
    Console.WriteLine("No Interface");
}
```

В блоке try последовательно подставляем следующие фрагменты кода.

Фрагмент 1:

```
// получим ссылку на интерфейс IDraw
IDraw idr = (IDraw)si;
idr.Draw();
```

Фрагмент 2:

```
// получим ссылку на интерфейс IDraw2
IDraw2 idr2 = (IDraw2)si;
idr2.Draw();
idr2.DrawToPrinter();
```

Фрагмент 2:

```
// не хотим использовать ссылку на интерфейс IDraw3
((IDraw3)si).Draw();
((IDraw3)si).DrawToPrinter();
((IDraw3)si).DrawToMetafile();
```

5.8. Контрольные вопросы и упражнения

1. Что означают this() и base() в списке инициализации конструктора?
2. Как описать абстрактный класс?
3. Как описать абстрактный метод?
4. Какой модификатор может иметь метод базового класса для того, чтобы метод производного класса мог иметь модификатор override?
5. Если абстрактный класс наследует интерфейс, какое определение метода интерфейса может быть в абстрактном классе?
6. Как описывается интерфейс?
7. Какие элементы может содержать интерфейс?
8. Сколько базовых классов и интерфейсов может иметь класс?
9. Как определить наличие интерфейса у объекта?
10. Как определяются методы с одинаковым названием, но принадлежащие разным интерфейсам?

6. Работа CS-106. Интерфейсы и коллекции C#

Цели:
- реализация системных интерфейсов.
Задачи:
- интерфейсы IEnumerable и IEnumerator;
- интерфейс ICloneable;
- интерфейс IComparable;
- интерфейс IComparer;
- класс ArrayList;

Опорные документы:

6.1. Интерфейсы перечисления

Создадим проект консольного приложения C#.
Название проекта — SharpNumerable.

Microsoft .NET объединяет в библиотеке базовых классов все лучшее, что накоплено в программировании за последние десятилетия. При изучении библиотеки базовых классов можно обнаружить множество классов-заготовок, использующих одни и те же стандартные интерфейсы. Следует научиться использовать эти интерфейсы, чтобы создавать свои классы.

Опишем класс Car (автомобиль):

```
class Car {
    private string name = "noname";
    public Car() { }
    public Car(string name) {
        this.name = name;
    }
    public string Name {
        get { return name; }
    }
}
```

Опишем класс Cars (свалка автомобилей):

```
class Cars {
    private Car[] cars;
    public Cars() {
        cars = new Car[4];
        cars[0] = new Car("alpha");
        cars[1] = new Car("porshe");
        cars[2] = new Car("nissan");
        cars[3] = new Car("ГАЗ-13");
    }
}
```

Было бы очень удобно использовать цикл foreach/in для просмотра свалки. А для этого нужно всего лишь описать интерфейс IEnumerable (перечисляемый).

Прежде необходимо удалить последнее слово Generic в using:

```
using System.Collections.Generic;
```

Должно получиться:

```
using System.Collections;
```

Описываем наследование класса IEnumerable:

```
class Cars : IEnumerable {
```

Тип IEnumerable имеет один метод GetEnumerator, возвращающий тип IEnumerator. И мы должны описать этот метод в классе Cars:

```
// IEnumerable
public IEnumerator GetEnumerator() {
    for (int i = 0; i < cars.Length; i++) {
        // многократный возврат
        yield return cars[i];
    }
}
```

Переходим в метод Main.

Создаем свалку автомобилей и просматриваем ее:

```
Cars carlot = new Cars();
foreach (Car c in carlot) {
    Console.WriteLine(c.Name);
}
```

При наследовании интерфейса IEnumerator нужно описать также его элементы: void Reset — перемещение к первому элементу (метод), bool MoveNext — перемещение к следующему элементу (метод), object Current — возвращает текущий элемент (свойство).

Описываем наследование IEnumerator:

```
class Cars : IEnumerable, IEnumerator {
```

Описываем индекс текущего элемента (в классе Cars):

```
private int current = -1;
```

Описываем реализацию метода Reset (в классе Cars):

```
// IEnumerator
public void Reset() {
    current = -1;
}
```

Описываем реализацию метода MoveNext (в классе Cars):

```
// IEnumerator
public bool MoveNext() {
    if (current < cars.Length - 1) {
        current++;
        return true;
    } else {
        return false;
    }
}
```

Описываем реализацию свойства Current (в классе Cars):

```
// IEnumerator
public object Current {
    get {
        return cars[current];
    }
}
```

В методе Main вызываем новые методы (после цикла foreach/in):

```
carlot.Reset();
carlot.MoveNext();
Console.WriteLine(((Car)carlot.Current).Name);
```

Кроме того, описав методы и свойства IEnumerator, мы можем значительно упростить метод GetEnumerator класса Cars:

```
// IEnumerable
public IEnumerator GetEnumerator() {
    return (IEnumerator)this;
    /*/
    for (int i = 0; i < cars.Length; i++) {
        // многократный возврат
        yield return cars[i];
    }
    /*/
}
```

6.2. Внутренний перечислитель

Создадим проект консольного приложения C#.

Название проекта — SharpNuminside.

Снова опишем класс Car:

```
class Car {
    private string name = "noname";
    public Car() { }
    public Car(string name) {
        this.name = name;
    }
    public string Name {
        get { return name; }
    }
}
```

Снова опишем класс Cars.

При этом создадим внутренний класс CarsEnumerator, как рекомендует MSDN 2005.

Изменять строки using не требуется.

Описание класса Cars:

```
class Cars : System.Collections.IEnumerable {
    private Car[] cars;
} // конец класса Cars
```

Конструктор класса Cars не изменился:

```
    public Cars() {
        cars = new Car[4];
        cars[0] = new Car("alpha");
        cars[1] = new Car("porshe");
        cars[2] = new Car("nissan");
        cars[3] = new Car("ГАЗ-13");
    }
```

Объявим закрытый класс CarsEnumerator, который будет выполнять роль перечислителя. Класс находится внутри класса Cars:

```
    private class CarsEnumerator : System.Collections.IEnumerator {
        private int current = -1;
        private Cars c;
    } // конец класса CarsEnumerator
```

Конструктор класса CarsEnumerator принимает ссылку на внешний класс:

```
        public CarsEnumerator(Cars c) {
            this.c = c;
        }
```

Остальная часть класса CarsEnumerator описывает необходимые элементы интерфейса IEnumerator:

```
        public void Reset() {
            current = -1;
        }
        public bool MoveNext() {
            if (current < c.cars.Length - 1) {
                current++;
                return true;
            } else {
                return false;
            }
        }
        public object Current {
            get { return c.cars[current]; }
        }
    } // конец класса CarsEnumerator
```

Наконец, описываем метод GetEnumerator класса Cars:

```
    public System.Collections.IEnumerator GetEnumerator() {
        return new CarsEnumerator(this);
    }
} // конец класса Cars
```

Убедимся, что проект не содержит ошибок.

В методе Main создаем свалку автомобилей carlot и описываем просмотр всех объектов:

```
    static void Main(string[] args) {
        Cars carlot = new Cars();
        foreach (object o in carlot) {
            Console.WriteLine(((Car)o).Name);
        }
    }
```

6.3. Клонируемые объекты

Создадим проект консольного приложения C#.
Название проекта — SharpClonable.
Опишем класс Point (точка на плоскости):

```
class Point {
    public int x, y;
    public Point() { }
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public override string ToString() {
        return "(" + x.ToString() + "," + y.ToString() + ")";
    }
}
```

В методе Main испытываем, как он работает:

```
    static void Main(string[] args) {
        Point p1 = new Point(2, 3);
        Console.WriteLine("1: " + p1.ToString());
    }
```

Теперь попробуем создать копию объекта Point:

```
        Point p2 = p1;
        p2.x = 3;
        Console.WriteLine("2: " + p2.ToString());
        Console.WriteLine("1: " + p1.ToString());
```

Поскольку объекты класса являются ссылочными типами, при использовании оператора = создается новая ссылка на тот же объект.

При этом используется метод MemberwizeClone по умолчанию.

Чтобы получить копию объекта вместо копии ссылки, нужно описать способ создания копии, используя интерфейс ICloneable.

Описываем класс Point2, наследующий интерфейс ICloneable:

```
class Point2 : ICloneable {
    public int x, y;
    public Point2() { }
    public Point2(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public override string ToString() {
        return "(" + x.ToString() + "," + y.ToString() + ")";
    }
}
```

Переопределяем единственный метод Clone интерфейса ICloneable:

```
    // ICloneable
    public object Clone() {
        return new Point2(this.x, this.y);
    }
```

Испытываем новый класс в методе Main:

```
Point2 p3 = new Point2(6, 6);
Point2 p4 = (Point2)p3.Clone();
p4.x = 3;
Console.WriteLine("4: " + p4.ToString());
Console.WriteLine("3: " + p3.ToString());
```

Убедимся, что в консоль выводится:

```
1: (2,3)
2: (3,3)
1: (3,3)
4: (3,6)
3: (6,6)
```

6.4. Сравниваемые объекты

Создадим проект консольного приложения C#.
Название проекта — SharpComparable.
Снова опишем класс Car:

```
class Car {
    private string name = "noname";
    public Car() { }
    public Car(string name) {
        this.name = name;
    }
    public string Name {
        get { return name; }
    }
}
```

В методе Main создаем и выводим список автомобилей:

```
    static void Main(string[] args) {
        Car[] cars = new Car[4];
        cars[0] = new Car("3");
        cars[1] = new Car("2");
        cars[2] = new Car("1");
        cars[3] = new Car("4");
        foreach (object c in cars) {
            Console.WriteLine(((Car)c).Name);
        }
    }
```

Пусть нам теперь требуется упорядочить список при помощи метода System.Array.Sort. Чтобы это было возможно сделать, объекты массива должны поддерживать интерфейс IComparable (сравниваемые).

Добавляем наследование интерфейса IComparable в класс Car:

```
class Car : IComparable {
```

Описываем в классе Car, как сравнивать объекты класса Car.
Мы используем сравнение объектов на основе имени:

```
    // IComparable
    int IComparable.CompareTo(object o) {
        return name.CompareTo(((Car)o).name);
    }
```

Теперь в методе Main мы можем отсортировать массив:

```
        Array.Sort(cars);
        Console.WriteLine("Sorted:");
        foreach (object c in cars) {
            Console.WriteLine(((Car)c).Name);
        }
```

Создадим проект консольного приложения C#.
Название проекта — SharpComparer.
Снова опишем класс Car:

```
class Car {
    private string name = "noname";
    public Car() { }
    public Car(string name) {
        this.name = name;
    }
    public string Name {
        get { return name; }
    }
}
```

Есть еще один интерфейс, используемый для сравнения объектов — интерфейс IComparer. Он предназначен для сравнивания каких-либо значений двух объектов, и состоит из метода Compare.

Описываем класс сравнивателя:

```
class SortByName : System.Collections.IComparer {
    public SortByName() { }
    // IComparer
    int System.Collections.IComparer.Compare(object a, object b) {
        return String.Compare(((Car)a).Name, ((Car)b).Name);
    }
}
```

Заметим, что данный класс ориентирован на сравнение типов Car.
В методе Main создаем массив автомобилей:

```
        Car[] cars = new Car[4];
        cars[0] = new Car("4");
        cars[1] = new Car("1");
        cars[2] = new Car("3");
        cars[3] = new Car("2");
```

и выводим список:

```
        foreach (Car c in cars) {
            Console.WriteLine(c.Name);
        }
```

Далее сортируем массив и выводим его:

```
Array.Sort(cars, new SortByName());
Console.WriteLine("Sorted:");
foreach (Car c in cars) {
    Console.WriteLine(c.Name);
}
```

Недостаток описанного подхода заключается в том, что при проектировании класса приходится учитывать внешние по отношению к нему методы. Было бы удобнее вместо этого использовать метод класса.

Метод Array.Sort имеет специально перегруженный вариант для этого случая. В классе Car нужно объявить статический метод, возвращающий IComparer:

```
public static System.Collections.IComparer SortByName {
    get {
        return (System.Collections.IComparer)new SortByName();
    }
}
```

Здесь new SortByName — представитель класса SortByName.

В методе Main теперь можно заменить вызов Sort:

```
//Array.Sort(cars, new SortByName());
Array.Sort(cars, Car.SortByName);
```

6.5. Коллекции C#

Создадим проект консольного приложения C#.

Название проекта — SharpCollection.

Снова опишем класс Car:

```
class Car {
    private string name = "noname";
    public Car() { }
    public Car(string name) {
        this.name = name;
    }
    public string Name {
        get { return name; }
    }
}
```

Встроенные в библиотеку базовых классов типы пространства имен System.Collections предназначены для работы с наборами элементов, и являются контейнерными классами. Интересны следующие интерфейсы:

Класс ICollection — определяет общие характеристики коллекций.

Класс IDictionary — представляет объект парами имя-значение.

Класс IList — добавление, удаление и индексирование элементов.

Кроме того, есть несколько готовых классов-коллекций, например:

Класс ArrayList — динамический массив объектов.

Класс Queue — стандартная очередь.

Класс SortedList — аналогичен словарю, но является индексируемым.

Класс Stack — стандартный стек.

Рассмотрим применение класса ArrayList. Сначала исправим using:

```
using System.Collections;
```

Описываем новый класс Cars:

```
class Cars : IEnumerable {
    private ArrayList cars;
    public Cars() {
        cars = new ArrayList();
    }
}
```

Добавим в класс Cars метод для добавления нового объекта:

```
    public void Add(Car c) {
        cars.Add(c);
    }
```

Следующее свойство возвращает количество объектов:

```
    public int Count {
        get {
            return cars.Count;
        }
    }
```

Следующий метод удаляет все объекты:

```
    public void Clear() {
        cars.Clear();
    }
```

Следующий метод удаляет элемент коллекции по его индексу:

```
    public void RemoveAt(int index) {
        cars.RemoveAt(index);
    }
```

Следующий метод проверяет наличие элемента в коллекции:

```
    public bool Contains(Car c) {
        return cars.Contains(c);
    }
```

Последний метод — метод перечисления интерфейса IEnumerable:

```
    // IEnumerable
    public IEnumerator GetEnumerator() {
        return cars.GetEnumerator();
    }
```

Тестируем класс в методе Main.
Сначала создаем представителя класса Cars и добавляем элементы:

```
    static void Main(string[] args) {
        Cars cars = new Cars();
        cars.Add(new Car("Alpha"));
        cars.Add(new Car("RR"));
        cars.Add(new Car("117"));
        cars.Add(new Car("Nissan"));

    }
```

Затем выводим список:

```
Console.WriteLine("We have {0} cars:", cars.Count);
foreach (Car c in cars) {
    Console.WriteLine(c.Name);
}
```

Удаляем третий элемент:

```
cars.RemoveAt(3);
Console.WriteLine("We have {0} cars:", cars.Count);
foreach (Car c in cars) {
    Console.WriteLine(c.Name);
}
```

Добавляем новый элемент:

```
Car a = new Car("Audi100");
cars.Add(a);
```

Проверяем наличие нового элемента:

```
if (cars.Contains(a)) {
    Console.WriteLine(a.Name + " is present");
}
```

Наконец, удаляем все:

```
cars.Clear();
Console.WriteLine("We have {0} cars.", cars.Count);
```

Интересный вопрос возникает — нельзя просто воспроизвести Cars от ArrayList? Можно, но тогда в класс можно будет добавлять произвольные объекты. В конце метода Main продемонстрируем использование ArrayList в таком качестве:

```
ArrayList ar = new ArrayList();
ar.Add(cars);
ar.Add(new Car("abc"));
ar.Add("Hi");
ar.Add(33);
foreach (object o in ar) {
    Console.WriteLine(o.ToString());
}
```

6.6. Контрольные вопросы и упражнения

1. Перечислите методы интерфейса IEnumerable.
2. Перечислите методы интерфейса IEnumerator.
3. Перечислите методы интерфейса ICloneable.
4. Перечислите методы интерфейса IComparable.
5. Перечислите методы интерфейса IComparer.
6. Перечислите методы класса ArrayList.