Словарь Dictionary<TKey, TValue>

В некоторых случаях может потребоваться использование пары «ключ-значение». В том случае, если в качестве ключа выступает обычное значение int, можно использовать обычный массив. Но иногда в качестве ключа необходимо использование значения какого-то нестандартного типа. В данном случае можно использовать тип Dictionary<TKey, TValue>, где и в качестве ключа, и в качестве значения можно использовать любые типы данных.

Создание словаря аналогично пройденному ранее созданию списка. Добавление в словарь происходит при помощи метода Add(TKey, TValue).

```
var lecturers = new Dictionary<string, string>();

lecturers.Add("Программирование", "Носовицкий В.Б.");
lecturers.Add("МатАн", "Петросян Н.С.");
//lecturers.Add("Программирование", "Шибаева А.Н."); // Ошибка
lecturers.Add("СиАОД", "Лакунина О.Н.");
lecturers.Add("ООП", "Кашуркин Н.");

foreach (var lect in lecturers)
    Console.WriteLine(lect);
Console.Write("Введи ключ для поиска:");
var key = Console.ReadLine();
string lecturer;
// Обращение по ключу
try
{// В случае, если элемент не будет найден, будет сгенерирована исключительная ситуация
    lecturer = lecturers[key];
}
catch
{
    lecturer = "Не найдено";
}
Console.WriteLine($"Результат обращения по ключу: {lecturer}");

// Поиск по ключу
var lecturer2 = lecturers.FirstOrDefault(l => l.Key.ToLower() == key.ToLower()).Value
?? "Не найдено";
Console.WriteLine($"Результат поиска по ключу: {lecturer2}");

if(lecturer != "Не найдено")
{
    Console.Write($"Введите новое значение для lecturer[{key}]: ");
    lecturers[key] = Console.ReadLine();

    foreach (var lect in lecturers)
        Console.WriteLine(lect);

    Console.WriteLine($"Удаление элемента с ключом {key}...");
    lecturers.Remove(key);
    foreach (var lect in lecturers)
        Console.WriteLine(lect);
}
```

```
[Программирование, Носовицкий В.Б.]
[МатАн, Петросян Н.С.]
[СиАОД, Лакунина О.Н.]
[ООП, Кашуркин Н.]
Введи ключ для поиска:МатАн
Результат обращения по ключу: Петросян Н.С.
Результат поиска по ключу: Петросян Н.С.
Введите новое значение для lecturer[МатАн]: Уварова Л.А.
[Программирование, Носовицкий В.Б.]
[МатАн, Уварова Л.А.]
[СиАОД, Лакунина О.Н.]
[ООП, Кашуркин Н.]
Удаление элемента с ключом МатАн...
[Программирование, Носовицкий В.Б.]
[СиАОД, Лакунина О.Н.]
[ООП, Кашуркин Н.]
```

Тип перечислений Enum

Тип перечисления – такой тип данных, который определяется набором перечисленных по имени констант. Enum относится к типам по значению (как структура).

```
public enum Имя
{
    Значение_1,
    Значение_2,
    Значение_3
}
```

Все значения констант по умолчанию имеют тип данных int, но вовсе не обязательно использовать именно его. Если необходимо использовать другой целочисленный тип, можно поставить после имени знак «:» и указать необходимый тип.

Все порядковые номера в энумераторе присваиваются значениям, начиная с единицы. Программист вправе сам решить значения для своего энумератора.

```
public enum MonthEnum
{
    January = 1,
    February = 2,
    March = 3,
    April = 4,
    May = 5,
    June = 6,
    July = 7,
    August = 8,
    September = 9,
    October = 10,
    November = 11,
    December = 12
}
```

Для использования энумератора необходимо вызвать его с синтаксисом: «Имя.Значение_1». Для получения значения энумератора можно использовать приведение типов ((int)Имя.Значение_1).

```
Console.WriteLine($"MonthEnum.January : {MonthEnum.January}"); // January
Console.WriteLine($"(int)MonthEnum.January : {(int)MonthEnum.January}"); // 1
```

Объектно-Ориентированное Программирование

Базовые понятия ООП

Базовыми понятиями объектно-ориентированного программирования являются объект и класс. Объект – это какой-то реально существующий предмет со всеми его индивидуальными характеристиками. Класс – это множество объектов с одинаковыми характеристиками и одинаковым поведением. При определении значений характеристик класс превращается в объект. Характеристики класса задают данными, а поведение – методами. В C# методы представляют собой функции; среди методов выделяют конструктор и финализатор – функции особого назначения и с особыми правилами оформления.

Основные принципы ООП

Основными принципами объектно-ориентированного программирования являются:

- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

Абстракция – выделение наиболее важных в рамках решения определенной задачи характеристик объекта. Например, необходимо написать программу по работе со студентами. В классе «Студент» будут описаны такие данные, как номер студенческого билета, фамилия, имя, дата рождения, место жительств и так далее. В рамках решения поставленной задачи мы абстрагируемся от таких характеристик, например, как рост и вес.
Инкапсуляция – объединение в одной структуре данных (структуре или классе) объявления данных и методов их обработки.
Наследование – возможность получать из одного класса другой с сохранением необходимых элементов в дочерних классах, модифицируя их, добавляя при необходимости новые возможности.
Полиморфизм – возможность иметь несколько реализаций одного элемента с автоматическим выбором подходящего.

Модификаторы доступа в языке C#

Модификаторы доступа позволяют определить доступность тех или иных элементов в структуре программы. В языке C# есть несколько модификаторов доступа:

- public – общедоступный элемент;
- private – закрытый элемент (элемент, помеченный таким модификатором доступа, доступен только в том элементе, в котором он объявлен);
- protected – элемент, доступный внутри того класса, в котором он объявлен, а так же в элементах, наследуемых от этого класса;
- internal – элемент, доступный для использования внутри сборки (проекта), в котором он указан;
- protected internal – элемент, доступный из текущей сборки и в наследниках от класса, в котором он объявлен;
- private protected – элемент, доступный внутри класса, в котором он объявлен, или в производных классах, которые определены в той же сборке.

Стоить отметить, что в языке C# в классах и структурах элементы по умолчанию имеют модификатор доступа private. Классы, написанные без модификатора доступа, по

умолчанию имеют модификатор доступа internal. Стоит также отметить, что действие модификатора доступа продолжается до знака «;».

```
class A // = internal class A
{
    int a; // Поле, доступное только внутри класса
    private int b; // Поле, доступное только внутри класса
    protected decimal c; // Поле, доступное внутри класса и классах-наследниках от
класса A
    public string d; // Поле, доступное в любом месте кода

    internal double e; // Поле, доступное в любом месте кода этой сборки

    protected internal int f; // Поле, доступное в любом месте кода этой сборки и в
классах-наследниках от A
    private protected char g; // Поле, доступное внутри класса A и в классах-
наследниках от A в этой сборке
}
```

Благодаря механизму модификаторов доступа можно скрывать некоторые методы и свойства класса от других частей программы, что и является инкапсуляцией.

Классы. Элементы класса

Класс предоставляет возможность программисту разрабатывать свои собственные типы данных путем группирования других типов и методов. Основным отличием классов от структур является возможность наследования.

Элементы класса:

- Поле. Переменная, инкапсулированная в классе. Как правило, закрытая.
- Свойство. Механизм для чтения, записи или вычисления значения элемента класса (например, скрытого). Позволяет легко получать доступ к данным и помогает повысить безопасность и гибкость методов. Метод доступа get используется для возврата значения свойства, а метод доступа set – для присвоения нового значения. Эти методы могут иметь различные уровни доступа. Параметр value используется в методе set для определения значения свойства и имеет тот же тип, что и свойство. Если set не реализован, то свойство доступно только для чтения.
- Метод. Механизм, позволяющий обрабатывать данные и возвращать результат работы над данными, если необходимо.
- Специальные методы и свойства класса.

```
public class B
{
    private decimal _seconds; // Поле
    public decimal Seconds // Свойство
    {
        get
        {
            return _seconds;
        }
        protected set
        {
            _seconds = value;
        }
    }

    public decimal Minutes // Свойство
    {
        get
        {
            return _seconds / 60;
        }
        set
        {
            _seconds = value * 60;
        }
    }

    public decimal GetHours() // Метод
    {
        var hours = _seconds / 3600;
        return hours;
    }
    public decimal GetHours2() => _seconds / 3600; // Метод

    public static void ConsolePrint(B b) // Статический метод
    {
        Console.WriteLine($"Секунд: {b.Seconds}");
        Console.WriteLine($"Минут: {b.Minutes}");
        Console.WriteLine($"Часов: {b.GetHours()}");
    }
}
```

```
B b = new B(); // Объект класса (Представитель класса, Instance)

//b.Seconds = 30; // Ошибка: Нет доступа
b.Minutes = 30; // set Свойства Minutes
Console.WriteLine(b.GetHours2()); // Вызов метода

//b.ConsolePrint(b); // Ошибка: Нет доступа
B.ConsolePrint(b); // Вызов статического элемента класса
```

Помимо стандартных элементов класса могут быть и статические элементы, которые будут относиться к самому классу, а не к его представителям. Примером использования статических элементов является использование методов класса Console (Console.WriteLine());

Конструктор и финализатор класса

При создании экземпляра любого класса вызывается специальный метод, называемый конструктором, который может быть использован для задания значений элементов класса. Конструктор представляет собой метод, который не имеет типа возвращаемого значения и имеет имя, такое же, как и имя класса, для которого объявлен данный конструктор. Класс или структура, могут иметь несколько конструкторов, которые должны отличаться между собой либо количеством принимаемых параметров, либо их типом.

```
public class C
{
    private int _x;
    private int _y;
    private readonly int _z; // Поле, недоступное для записи вне объявления и
конструктора
    private int k;

    public C() // Конструктор без параметров
    {
        _x = 100500;
        _z = -100500;
    }
    public C(int x, int y, int z, int k) // Конструктор с параметрами
    {
        _x = x;
        _y = y;
        _z = z;
        this.k = k; // this.k - k, принадлежащее классу C
    }
    public C(C c) // Конструктор копирования
    {
        _x = c._x;
        _y = c._y;
        _z = c._z;
        k = c.k;
    }
    public void UpdateZ(int z)
    {
        //_z = z; // Ошибка: Поле ReadOnly
    }
    ~C() // Финализатор класса
    {
        Console.WriteLine("Вызван финализатор класса C");
        throw new Exception("Вызван финализатор класса C");
    }
}
```

Если конструктор не будет объявлен в явном виде, то среда сгенерирует конструктор без параметров. За крайне редким исключением конструктору класса должен быть присвоен модификатор доступа public.

В общем случае в C# разрешено присвоение между объектами одного и то же класса. На практике это означает, что мы получим два указателя на один и тот же объект. Для обеспечения создания нового объекта, которому в момент создания были переданы значения данных существующего объекта, но при этом под него выделялась собственная область памяти и в дальнейшем эти два объекта были бы полностью независимыми, необходим конструктор копирования. Единственным формальным параметром конструктора копирования всегда является переменная типа копируемый класс. При наличии конструктора копирования в классе всегда должен быть и обычный конструктор.

За выделение и освобождение памяти в языке C# отвечает сборщик мусора. При каждом создании нового объекта среда выполнения выделяет память для объекта из управляемой динамически распределяемой памяти (чаще всего называемой просто кучей). Когда количество доступной для выделения памяти в куче достигает определенного критического значения, сборщик мусора «отыскивает» в ней объекты, которые больше не используются, и освобождает занимаемую ими память.

Финализатор – метод класса, который автоматически вызывается средой выполнения в промежутке времени между моментом, когда объект этого класса был помечен сборщиком мусора, как неиспользуемый, и моментом удаления объекта. Соответственно, мы не можем с полной уверенностью сказать, когда данный метод отработает, в отличии от деструктора в языке C++ (деструктор вызывается при выходе из области видимости объекта класса или при явном освобождении занимаемой памяти). Финализатор в C# представляет собой реализацию метода Finalize(), который нельзя переопределить, поэтому для вызова финализатора используется синтаксис вызова деструктора из C++, и только в момент компиляции компилятор называет его Finalize().

Наследование

В C# допускается простое наследование: каждый класс может иметь только одного предка. Используя наследование, можно создать базовый класс, который определяет характеристики, присущие множеству связанных объектов. Этот класс затем может быть унаследован другими классами с добавлением в каждый из них своих особенностей. Равнозначные термины (пары): базовый класс – класс-наследник; родительский класс – дочерний класс; класс предок – класс-наследник.

Создадим в качестве примера базовый класс для обработки массива, включающий определение массива, его ввод и вывод. К элементам базового класса с атрибутом доступа private нет доступа из классов – наследников, они, таким образом, не наследуются. Поэтому рекомендуют (если нет на этот счет особых соображений) дать элементам базового класса атрибут доступа protected.

```
class MyBaseArray
    {
        protected int[] array;

        /// <summary>
        /// Конструктор класса MyBaseArray без параметров
        /// </summary>
        public MyBaseArray()
```

```
        {
            Console.Write("Количество элементов = ");
            int size = Convert.ToInt32(Console.ReadLine());
            array = new int[size];
            for(int i = 0; i< size; i++)
            {
                Console.Write("array[" + i + "] = ");
                array[i] = Convert.ToInt32(Console.ReadLine());
            }
        }

        /// <summary>
        /// Конструктор класса MyBaseArray с одним параметром
        /// </summary>
        /// <param name="size">Размер массива</param>
        public MyBaseArray(int size)
        {
            array = new int[size];
            for (int i = 0; i < size; i++)
            {
                Console.Write("array[" + i + "] = ");
                array[i] = Convert.ToInt32(Console.ReadLine());
            }
        }

        /// <summary>
        /// Функция вывода массива на экран
        /// </summary>
        public void output()
        {
            Console.WriteLine();
            for (int i = 0; i < array.Length; i++)
            {
                Console.WriteLine("array[" + i + "] = " + array[i]);
            }
        }
    }
```

На его базе можно построить классы обработки массивов. В нашем случае – нахождение суммы. Класс-наследник включает все данные своего предка (за исключением данных с атрибутом доступа private). Наследуются по общим правилам и индексаторы, и свойства, а также методы перегрузки операторов.

```
class ProcessArray: MyBaseArray // В качестве базового класса выбран класс MyBaseArray
    {
        int _q;

        /// <summary>
        /// Конструктор класса ProcessArray (наследника)
        /// </summary>
        public ProcessArray()
        {
            Console.Write("Граница: ");
            _q = Convert.ToInt32(Console.ReadLine());
        }

        /// <summary>
        /// Функция вычисления суммы элементов массива
        /// </summary>
        /// <returns> Сумма элементов массива, больших заданного значения </returns>
        public int Sum()
        {
            int s = 0;
            foreach (var item in array.Where(i => i > _q))
            {
```

```
                s += item;
            }
            return s;
        }
    }
```

Использование созданных классов

```
class Program
    {
        static void Main(string[] args)
        {
            ProcessArray myFirstInheritance = new ProcessArray();
                            // Создание экземпляра класса - наследника
            myFirstInheritance.Output(); // Обращение к методу предка
            int mySum = myFirstInheritance.Sum(); // Обращение к собственному методу

            Console.WriteLine("Сумма = " + mySum);
            Console.ReadLine();
        }
    }
```

При создании экземпляра класса, имеющего предка, запускаются все конструкторы: в первую очередь конструктор базового класса и затем конструктор класса – наследника. В нашем случае это означает, что будет осуществлен ввод сначала массива и вслед за ним – границы. При наличии большего количества уровней наследования подряд будут запущены конструкторы всех уровней иерархии, начиная с базового.

Если конструкторы не имеют формальных параметров, то при этом никаких проблем не возникает: каждый конструктор независимо от других выполняет свои операторы. Осталось решить вопрос: как обеспечить передачу параметра/ параметров конструктору класса – предка, в нашем случае конструктору 2. Проще всего это выполнить с помощью списка инициализации в конструкторе класса – наследника.

```
/// <summary>
/// Конструктор класса ProcessArray (наследника) с параметром
/// </summary>
/// <param name="size">Фактический параметр для конструктора-предка</param>
/// <param name="q">Граница</param>
public ProcessArray(int size, int q): base(size)
{
     _q = q;
     Console.Write("Граница: " + _q);
}
```

Запись base(size) означает, что конструктору базового класса в качестве фактического параметра будет передано значение size.

```
static void Main(string[] args)
{
    Console.Write("Количество элементов: ");
    int size = Convert.ToInt32(Console.ReadLine());
    Console.Write("Минимальная граница для рассчета суммы: ");
    int minLimit = Convert.ToInt32(Console.ReadLine());
    var array = new ProcessArray(size, minLimit);
    array.Output(); // Метод класса MyBaseArray
    // Метод класса ProcessArray
    Console.WriteLine($"Сумма элементов массива, значения которых больше {minLimit},
равна {array.Sum()}");

    Console.ReadLine();
}
```

Лучше всего придерживаться следующего правила: при написании конструктора класса – наследника позаботиться о параметрах непосредственного предка. Таким образом, даже при большом количестве уровней иерархии будет обеспечена согласованная работа конструкторов.