Л а б о р а т о р н а я р а б о т а № 10

**ПОЛИНОМЫ В WINDOWS FORMS. КЛАССЫ, ИХ ПОЛЯ И МЕТОДЫ.** [[**РАБОТА С МАССИВАМИ**](#_Toc290318461)**.**](#_Toc290318445)

**Цель работы:**

* ознакомиться с принципами создания классов;
* научиться применять основные принципы пользовательского интерфейса;
* получить навыки в организации различных типов табличных элементов.

**Краткие теоретические сведения.**

## Классы и объекты

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

В принципе ранее уже использовались классы. Например, тип **string**, который представляет строку, фактически является классом. Или, например, класс **Console**, у которого метод WriteLine() выводит на консоль некоторую информацию. Теперь же посмотрим, как мы можем определять свои собственные классы.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова **сlass**:

|  |
| --- |
| class название\_класса  {      // содержимое класса  } |

После слова **class** идет имя класса и далее в фигурных скобках идет собственно содержимое класса. Например, определим в файле **Program.cs** класс Person, который будет представлять человека:

|  |
| --- |
| class Person  {    } |

Начиная с версии C# 12, если класс имеет пустое определение, то фигурные скобки после названия типа можно не использовать:

|  |
| --- |
| class Person; |

Однако такой класс не особо показателен, поэтому добавим в него некоторую функциональность.

## Поля и методы класса

Класс может хранить некоторые данные. Для хранения данных в классе применяются **поля**. По сути **поля класса** - это переменные, определенные на уровне класса.

Кроме того, класс может определять некоторое поведение или выполняемые действия. Для определения поведения в классе применяются методы.

Итак, добавим в класс Person поля и методы:

|  |
| --- |
| class Person  {      public string name = "Undefined";   // имя      public int age;                     // возраст        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

В данном случае в классе Person определено поле name, которое хранит имя, и поле age, которое хранит возраст человека. В отличие от переменных, определенных в методах, поля класса могут иметь модификаторы, которые указываются перед полем. Так, в данном случае, чтобы все поля были доступны вне класса Person поля определены с модификатором **public**.

При определении полей мы можем присвоить им некоторые значения, как в примере выше в случае переменной name. Если поля класса не инициализированы, то они получают значения по умолчанию. Для переменных числовых типов это число 0.

Также в классе Person определен метод Print(). Методы класса имеют доступ к его поля, и в данном случае обращаемся к полям класса name и age для вывода их значения на консоль. И чтобы этот метод был виден вне класса, он также определен с модификатором **public**.

## Создание объекта класса

После определения класса мы можем создавать его объекты. Для создания объекта применяются **конструкторы**. По сути конструкторы представляют специальные методы, которые называются так же как и класс, и которые вызываются при создании нового объекта класса и выполняют инициализацию объекта. Общий синтаксис вызова конструктора:

|  |
| --- |
| new конструктор\_класса(параметры\_конструктора); |

Сначала идет оператор **new**, который выделяет память для объекта, а после него идет вызов **конструктора**.

## Конструктор по умолчанию

Если в классе не определено ни одного конструктора (как в случае с нашим классом Person), то для этого класса автоматически создается пустой конструктор по умолчанию, который не принимает никаких параметров.

Теперь создадим объект класса Person:

|  |
| --- |
| Person tom = new Person();  // создание объекта класса Person    // определение класса Person  class Person  {      public string name = "Undefined";      public int age;        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

Для создания объекта Person используется выражение new Person(). В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект, и через эту переменную мы можем использовать данный объект и обращаться к его функциональности.

## Обращение к функциональности класса

Для обращения к функциональности класса - полям, методам (а также другим элементам класса) применяется точечная нотация точки - после объекта класса ставится точка, а затем элемент класса:

|  |
| --- |
| объект.поле\_класса  объект.метод\_класса(параметры\_метода) |

Например, обратимся к полям и методам объекта Person:

|  |
| --- |
| Person tom = new Person();  // создание объекта класса Person    // Получаем значение полей в переменные  string personName = tom.name;  int personAge = tom.age;  Console.WriteLine($"Имя: {personName}  Возраст {personAge}");   // Имя: Undefined  Возраст: 0    // устанавливаем новые значения полей  tom.name = "Tom";  tom.age = 37;    // обращаемся к методу Print  tom.Print();    // Имя: Tom  Возраст: 37    class Person  {      public string name = "Undefined";      public int age;        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

## Создание конструкторов

Выше для создания объекта использовался конструктор по умолчанию. Однако мы сами можем определить свои конструкторы. Как правило, конструктор выполняет инициализацию объекта. При этом если в классе определяются свои конструкторы, то он лишается конструктора по умолчанию.

На уровне кода конструктор представляет метод, который называется по имени класса, который может иметь параметры, но для него не надо определять возвращаемый тип. Например, определим в классе Person простейший конструктор:

|  |
| --- |
| Person tom = new Person();  // Создание объекта класса Person    tom.Print();    // Имя: Tom  Возраст: 37    class Person  {      public string name;      public int age;      public Person()      {          Console.WriteLine("Создание объекта Person");          name = "Tom";          age = 37;      }      public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

Итак, здесь определен конструктор, который выводит на консоль некоторое сообщение и инициализирует поля класса.

|  |
| --- |
| public Person()  {      Console.WriteLine("Создание объекта Person");      name = "Tom";      age = 37;  } |

Конструкторы могут иметь модификаторы, которые указываются перед именем конструктора. Так, в данном случае, чтобы конструктор был доступен вне класса Person, он определен с модификатором **public**.

Определив конструктор, мы можем вызвать его для создания объекта Person:

|  |
| --- |
| Person tom = new Person();  // Создание объекта Person |

В данном случае выражение Person() как раз представляет вызов определенного в классе конструктора (это больше не автоматический конструктор по умолчанию, которого у класса теперь нет). Соответственно при его выполнении на консоли будет выводиться строка "Создание объекта Person"

Подобным образом мы можем определять и другие конструкторы в классе. Например, изменим класс Person следующим образом:

|  |
| --- |
| Person tom = new Person();          // вызов 1-ого конструктора без параметров  Person bob = new Person("Bob");     //вызов 2-ого конструктора с одним параметром  Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами    tom.Print();          // Имя: Неизвестно  Возраст: 18  bob.Print();          // Имя: Bob  Возраст: 18  sam.Print();          // Имя: Sam  Возраст: 25    class Person  {      public string name;      public int age;      public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор      public Person(string n) { name = n; age = 18; }         // 2 конструктор      public Person(string n, int a) { name = n; age = a; }   // 3 конструктор        public void Print()      {          Console.WriteLine($"Имя: {name}  Возраст: {age}");      }  } |

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. И мы можем вызвать один из этих конструкторов для создания объекта класса.

Консольный вывод данной программы:

Имя: Неизвестно Возраст: 18

Имя: Bob Возраст: 18

Имя: Sam Возраст: 25

Стоит отметить, что начиная с версии C# 9 мы можем сократить вызов конструктора, убрав из него название типа:

|  |
| --- |
| Person tom = new ();            // аналогично new Person();  Person bob = new ("Bob");       // аналогично new Person("Bob");  Person sam = new ("Sam", 25);   // аналогично new Person("Sam", 25); |

### Ключевое слово this

Ключевое слово **this** представляет ссылку на текущий экземпляр/объект класса. В каких ситуациях оно нам может пригодиться?

|  |
| --- |
| Person sam = new("Sam", 25);  sam.Print();          // Имя: Sam  Возраст: 25    class Person  {      public string name;      public int age;      public Person() { name = "Неизвестно"; age = 18; }      public Person(string name) { this.name = name; age = 18; }      public Person(string name, int age)      {          this.name = name;          this.age = age;      }      public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");  } |

В примере выше во втором и третьем конструкторе параметры называются также, как и поля класса. И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово **this**. Так, в выражении

**this.name = name;**

первая часть - this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово **this** можно обращаться к любому полю или методу.

## Первичные конструкторы

Начиная с версии **C# 12** в язык C# была добавлена поддержка **первичных конструкторов** (Primary constructors). Первичные конструкторы позволяют добавлять параметры к определению класса и использовать эти параметры внутри класса:

|  |
| --- |
| var tom = new Person("Tom", 38);  Console.WriteLine(tom);    public class Person(string name, int age)  {      public Person(string name) : this(name, 18) { }      public void Print() => Console.WriteLine($"name: {name}, age: {age}");  } |

Здесь для класса Person определен первичный конструктор с двумя параметрами - name и age. Эти параметры применяются для используются в методе Print.

За кадром для каждого параметра первичного конструктора в классе создается приватное поле, которое хранит значение параметра. Благодаря этому они могут использоваться в теле класса.

Кроме первичных конструкторов класс может определять дополнительные конструкторы, как примере выше. Но эти дополнительные конструкторы должны вызывать первичный конструктор:

|  |
| --- |
| public Person(string name) : this(name, 18) { } |

## Инициализаторы объектов

Для инициализации объектов классов можно применять **инициализаторы**. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

|  |  |
| --- | --- |
| 1  2  3  4 | Person tom = new Person { name = "Tom", age = 31 };  // или так  // Person tom = new() { name = "Tom", age = 31 };  tom.Print();          // Имя: Tom  Возраст: 31 |

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания. При использовании инициализаторов следует учитывать следующие моменты:

* С помощью инициализатора мы можем установить значения только доступных из вне класса полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.
* Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

Инициализаторы удобно применять, когда поле или свойство класса представляет другой класс:

|  |
| --- |
| Person tom = new Person{ name = "Tom", company = { title = "Microsoft"} };  tom.Print();          // Имя: Tom  Компания: Microsoft    class Person  {      public string name;      public Company company;      public Person()      {          name = "Undefined";          company = new Company();      }      public void Print() => Console.WriteLine($"Имя: {name}  Компания: {company.title}");  }    class Company  {      public string title = "Unknown";  } |

Обратите внимание, как устанавливается поле company:

|  |
| --- |
| company = { title = "Microsoft"} |

## Свойства

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют **свойства**. Они обеспечивают простой доступ к полям классов и структур, узнать их значение или выполнить их установку.

## Определение свойств

Стандартное описание свойства имеет следующий синтаксис:

|  |
| --- |
| [модификаторы] тип\_свойства название\_свойства  {      get { действия, выполняемые при получении значения свойства}      set { действия, выполняемые при установке значения свойства}  } |

Вначале определения свойства могут идти различные модификаторы, в частности, модификаторы доступа. Затем указывается тип свойства, после которого идет название свойства. Полное определение свойства содержит два блока: **get** и **set**.

В блоке **get** выполняются действия по получению значения свойства. В этом блоке с помощью оператора **return** возвращаем некоторое значение.

В блоке **set** устанавливается значение свойства. В этом блоке с помощью параметра **value** мы можем получить значение, которое передано свойству.

Блоки **get** и **set** еще называются акссесорами или методами доступа (к значению свойства), а также геттером и сеттером.

Рассмотрим пример:

|  |
| --- |
| Person person = new Person();    // Устанавливаем свойство - срабатывает блок Set  // значение "Tom" и есть передаваемое в свойство value  person.Name = "Tom";    // Получаем значение свойства и присваиваем его переменной - срабатывает блок Get  string personName = person.Name;  Console.WriteLine(personName);  // Tom    class Person  {      private string name = "Undefined";        public string Name      {          get          {              return name;    // возвращаем значение свойства          }          set          {              name = value;   // устанавливаем новое значение свойства          }      }  } |

Здесь в классе Person определено приватное поле name, которая хранит имя пользователя, и есть общедоступное свойство Name. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной name. В свойстве в блоке **get** возвращаем значение поля:

|  |
| --- |
| get { return name; } |

А в блоке **set** устанавливаем значение переменной name. Параметр **value** представляет передаваемое значение, которое передается переменной name.

|  |
| --- |
| set { name = value; } |

В программе мы можем обращаться к этому свойству, как к обычному полю. Если мы ему присваиваем какое-нибудь значение, то срабатывает блок **set**, а передаваемое значение передается в параметр **value**:

|  |
| --- |
| person.Name = "Tom"; |

Если мы получаем значение свойства, то срабатывает блок **get**, который по сути возвращает значение переменной name:

|  |
| --- |
| string personName = person.Name; |

То есть по сути свойство Name ничего не хранит, оно выступает в роли посредника между внешним кодом и переменной name.

Возможно, может возникнуть вопрос, зачем нужны свойства, если мы можем в данной ситуации обходиться обычными полями класса? Но свойства позволяют вложить дополнительную логику, которая может быть необходима при установке или получении значения. Например, нам надо установить проверку по возрасту:

|  |
| --- |
| Person person = new Person();    Console.WriteLine(person.Age);  // 1  // изменяем значение свойства  person.Age = 37;  Console.WriteLine(person.Age);  // 37  // пробуем передать недопустимое значение  person.Age = -23;               // Возраст должен быть в диапазоне от 1 до 120  Console.WriteLine(person.Age);  // 37 - возраст не изменился    class Person  {      int age = 1;      public int Age      {          set          {              if (value < 1 || value > 120)                  Console.WriteLine("Возраст должен быть в диапазоне от 1 до 120");              else                  age = value;          }          get { return age; }      }  } |

В данном случае переменная age хранит возраст пользователя. Напрямую мы не можем обратиться к этой переменной - только через свойство Age. Причем в блоке **set** мы устанавливаем значение, если оно соответствует некоторому разумному диапазону. Поэтому при передаче свойству Age значения, которое не входит в этот диапазон, значение переменной не будет изменяться:

|  |
| --- |
| person.Age = -23; |

## Автоматические свойства

Свойства управляют доступом к полям класса. Однако что, если у нас с десяток и более полей, то определять каждое поле и писать для него однотипное свойство было бы утомительно. Поэтому в .NET были добавлены автоматические свойства. Они имеют сокращенное объявление:

|  |
| --- |
| class Person  {      public string Name { get; set; }      public int Age { get; set; }        public Person(string name, int age)      {          Name = name;          Age = age;      }  } |

На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует при компиляции.

В чем преимущество автосвойств, если по сути они просто обращаются к автоматически создаваемой переменной, почему бы напрямую не обратиться к переменной без автосвойств? Дело в том, что в любой момент времени при необходимости мы можем развернуть автосвойство в обычное свойство, добавить в него какую-то определенную логику.

Стоит учитывать, что нельзя создать автоматическое свойство только для записи, как в случае со стандартными свойствами.

Автосвойствам можно присвоить значения по умолчанию (инициализация автосвойств):

|  |
| --- |
| Person tom = new();    Console.WriteLine(tom.Name);    // Tom  Console.WriteLine(tom.Age);    // 37    class Person  {      public string Name { get; set; } = "Tom";      public int Age { get; set; } = 37;  } |

И если мы не укажем для объекта Person значения свойств Name и Age, то будут действовать значения по умолчанию.

Автосвойства также могут иметь модификаторы доступа:

|  |
| --- |
| class Person  {      public string Name { private set; get;}      public Person(string name) => Name = name;  } |

Мы можем убрать блок set и сделать автосвойство доступным только для чтения. В этом случае для хранения значения этого свойства для него неявно будет создаваться поле с модификатором readonly, поэтому следует учитывать, что подобные get-свойства можно установить либо из конструктора класса, как в примере выше, либо при инициализации свойства:

|  |
| --- |
| class Person  {      // через инициализацию свойства      public string Name { get; } = "Tom";      // через конструктор      public Person(string name) => Name = name;  } |

## Сокращенная запись свойств

Как и методы, мы можем сокращать определения свойств. Поскольку блоки **get** и **set** представляют специальные методы, то как и обычные методы, если они содержат одну инструкцию, то мы их можем сократить с помощью оператора **=>**:

|  |
| --- |
| class Person  {      string name;      public string Name      {          get => name;          set => name = value;      }  } |

Также можно сокращать все свойство в целом:

|  |
| --- |
| class Person  {      string name;        // эквивалентно public string Name { get { return name; } }      public string Name => name;  } |

## Перегрузка методов

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еще называется **перегрузкой методов** (method overloading).

И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем, но разной сигнатурой. Что такое сигнатура? **Сигнатура** складывается из следующих аспектов:

* Имя метода
* Количество параметров
* Типы параметров
* Порядок параметров
* Модификаторы параметров

Но названия параметров в сигнатуру **НЕ** входят. Например, возьмем следующий метод:

|  |
| --- |
| public int Sum(int x, int y)  {      return x + y;  } |

У данного метода сигнатура будет выглядеть так: Sum(int, int)

И перегрузка метода как раз заключается в том, что методы имеют разную сигнатуру, в которой совпадает только название метода. То есть методы должны отличаться по:

* Количеству параметров
* Типу параметров
* Порядку параметров
* Модификаторам параметров

Например, пусть у нас есть следующий класс:

|  |
| --- |
| class Calculator  {      public void Add(int a, int b)      {          int result = a + b;          Console.WriteLine($"Result is {result}");      }      public void Add(int a, int b, int c)      {          int result = a + b + c;          Console.WriteLine($"Result is {result}");      }      public int Add(int a, int b, int c, int d)      {          int result = a + b + c + d;          Console.WriteLine($"Result is {result}");          return result;      }      public void Add(double a, double b)      {          double result = a + b;          Console.WriteLine($"Result is {result}");      }  } |

Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.

Первые три версии метода отличаются по количеству параметров. Четвертая версия совпадает с первой по количеству параметров, но отличается по их типу. При этом достаточно, чтобы хотя бы один параметр отличался по типу. Поэтому это тоже допустимая перегрузка метода Add.

То есть мы можем представить сигнатуры данных методов следующим образом:

|  |
| --- |
| Add(int, int)  Add(int, int, int)  Add(int, int, int, int)  Add(double, double) |

После определения перегруженных версий мы можем использовать их в программе:

|  |
| --- |
| Calculator calc = new Calculator();  calc.Add(1, 2); // 3  calc.Add(1, 2, 3); // 6  calc.Add(1, 2, 3, 4); // 10  calc.Add(1.4, 2.5); // 3.9 |

Также перегружаемые методы могут отличаться по используемым модификаторам. Например:

|  |
| --- |
| void Increment(ref int val)  {      val++;      Console.WriteLine(val);  }    void Increment(int val)  {      val++;      Console.WriteLine(val);  } |

В данном случае обе версии метода Increment имеют одинаковый набор параметров одинакового типа, однако в первом случае параметр имеет модификатор ref. Поэтому обе версии метода будут корректными перегрузками метода Increment.

А отличие методов по возвращаемому типу или по имени параметров не является основанием для перегрузки. Например, возьмем следующий набор методов:

|  |
| --- |
| int Sum(int x, int y)  {      return x + y;  }  int Sum(int number1, int number2)  {      return number1 + number2;  }  void Sum(int x, int y)  {      Console.WriteLine(x + y);  } |

Сигнатура у всех этих методов будет совпадать:

|  |
| --- |
| Sum(int, int) |

Поэтому данный набор методов не представляет корректные перегрузки метода Sum и **работать не будет**.

**Практическая часть.**

Построить приложение Windows Forms:

* Первичный вид окна представляет собой лишь поле для ввода степени полинома и кнопку построения шаблонов для полинома.
* В качестве шаблонов использовать массив TextBox-ов или компонент GridViev на выбор.
* Результирующие полиномы выводятся в свободной форме.

Поле для ввода степни полинома и кнопка построения шаблонов должны быть заблокированы. После нажатия кнопки выполнения задания в окне не должно быть возможности для редактирования каких-либо компонентов, но должна присутствовать кнопка сброса, приводящая окно к первоначальному виду.

Создать класс «Полином», полями которого являются степень полинома и коэффициенты или степень полинома, первый коэффициент и его корни (в зависимости задания по варианту). В классе предусмотреть методы для ввода полинома и вывода. Все задания варианта реализовать методами.

## Задание.

## Полиномы: вычисление значений, преобразования

1. Полином P(x) задан своими корнями и старшим коэффициентом an. Дан массив координат X. Вычислить массив значений полинома в точках xi.
2. (задача интерполяции) Полином P(x) задан координатами n+1 точек, через которые он проходит. Дан массив координат X. Вычислить массив значений полинома в точках xi.
3. Полином P(x) задан своими корнями и старшим коэффициентом an. Вычислить коэффициенты полинома.
4. Полином P(x) задан своими коэффициентами. Дан массив чисел X. Построить полином Q(X), имеющий своими корнями числа из массива X и корни полинома P(x).
5. Полином P(x) задан своими коэффициентами. Для полинома известны два его корня – x0 и xn. Построить полином Q(x), корни которого совпадают с корнями полинома P(x) за исключением корней x0 и xn.
6. Полиномы P(x) и Q(x) заданы своими корнями и старшими коэффициентами. Вычислить коэффициенты суммы полиномов P(x) и Q(x).
7. Полиномы P(x) и Q(x) заданы своими корнями и старшими коэффициентами. Вычислить коэффициенты произведения полиномов P(x) и Q(x).
8. Полиномы P(x) и Q(x) заданы своими коэффициентами. Вычислить коэффициенты суммы полиномов P(x) и Q(x).
9. Полиномы P(x) и Q(x) заданы своими коэффициентами. Вычислить коэффициенты произведения полиномов P(x) и Q(x).
10. Полиномы P(x) и Q(x) заданы точками, через которые они проходят. Вычислить коэффициенты суммы полиномов P(x) и Q(x).
11. Полиномы P(x) и Q(x) заданы точками, через которые они проходят. Вычислить коэффициенты произведения полиномов P(x) и Q(x).
12. **Контрольные вопросы** 
    * + 1. Как определяется класс?
        2. Для чего в классе нужны поля?
        3. Для чего в классе нужны методы?
        4. Как создать объект класса?
        5. Как обратиться к функциональности класса?
        6. Какие модификаторы имеют конструкторы класса?
        7. Что такое первичные констркуторы?
        8. Какой синтаксис имеет стандартное описание свойства
        9. Что такое автоматическое свойство?
        10. Как выглядит сокращенная запись свойств?
        11. Что такое сигнатура метода?
        12. Что такое перегрузка метода?