МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«Владимирский государственный университет имени Александра Григорьевича и Николая Григорьевича Столетовых» (ВлГУ)

Методические указания для выполнения

лабораторных работ по дисциплине

«Объектно-ориентированный анализ и программирование»

Составитель:

Виноградов Д.В.

Владимир - 2014

УДК 004.432 + 004.421.2 (076.5)

ББК 22.18 я73

 В3

Рецензент –

**Виноградов Д.В.**

|  |  |
| --- | --- |
| **В3** | Объектно-ориентированное программирование: методические указания для выполнения практических работ / Виноградов Д.В./ Владимирский гос. ун-т (ВлГУ). – Владимир: (ВлГУ), 2014. – 80 с. |
|  | Пособие предназначено для проведения лабораторных работ по курсу «Объектно-ориентированное программирование» студентов кафедры «Бизнес-информатика и экономика» ГОУ ВПО ВлГУ по направлению 080500.62 «Бизнес-информатика». |

УДК 004.432 + 004.421.2 (076.5)

ББК 22.18 я73

 ©Владимирский государственный университет, 2014

 ©Виноградов Д.В., 2014

**Лабораторная работа №1. Классы и объекты: общие сведения, поля, константы, методы**

**Цель работы:**

Познакомиться с основой объектного подхода в языке C#, созданием объектов и классов.

**Теория**

С теоретической точки зрения: класс – это тип, описывающий устройство объектов; поля – это переменные, принадлежащие классу; методы – это функции (процедуры), принадлежащие классу; объект – это экземпляр класса, сущность в адресном пространстве компьютера.

Можно сказать, что класс является шаблоном для объекта, описывающим его структуру и поведение. Поля класса определяют структуру объекта, методы класса – поведение объекта, значение полей объекта – состояние объекта.

С точки зрения практической реализации (в самом тексте программы) класс является типом данных, а объект – переменной этого типа.

Методы и переменные, составляющие класс, называются членами класса. При определении класса объявляются данные, которые он содержит, и код, работающий с этими данными. Данные содержатся в переменных экземпляра, которые определены классом, а код содержится в методах. В С# определены несколько специфических разновидностей членов класса. Это — поля, константы, методы, конструкторы, деструкторы, индексаторы, события, операторы, свойства и вложенные типы.

Классы объявляются с помощью ключевого слова class:

[уровень доступа] **class** <имя класса>

{

// Описание полей

// Описание методов

// Описание прочих членов класса

}

Уровней доступа для классов два: 1) public – доступ к классу возможен из любого места одной сборки либо из другой сборки[[1]](#footnote-1), на которую есть ссылка; 2) internal – доступ к классу возможен только из сборки, в которой он объявлен.

При объявлении класса модификатор доступа можно не указывать, при этом будет применяться режим по умолчанию internal.

Класс следует объявлять внутри пространства имен namespace, но за пределами другого класса (исключение – вложенные типы).

Классы содержат поля данных и объявления пользовательских процедур и функций, предназначенные для выполнения над объектами различных операций и которые обобщенно называют методами

**Поле** – это переменная любого типа, которая объявлена непосредственно в классе. Описание полей делается следующим образом:

<уровень доступа> <тип переменной> <имя переменной>;

**Константа –** неизменные значения, известные во время компиляции и неизменяемые на протяжении времени существования программы. Описание и инициализация констант делается следующим образом:

<уровень доступа>**const** <тип константы><имя константы>=<значение константы>;

**Метод** представляет собой блок кода, содержащий набор инструкций, реализующих поведение экземпляров класса. В C# все инструкции выполняются в коде методов (в том числе метод Main, который является точкой входа для каждого приложения C# и вызывается при запуске программы).

Описание метода осуществляется следующим образом:

<уровень доступа> <возвращаемый тип><имя метода > (<список параметров>){тело\_метода}

Для членов класса уровней доступа пять: 1) public – доступ к члену возможен из любого места одной сборки, либо из другой сборки, на которую есть ссылка; 2) protected – доступ к члену возможен только внутри класса, либо в классе-наследнике (при наследовании); 3) internal – доступ к члену возможен только из сборки, в которой он объявлен; 4) private – доступ к члену возможен только внутри класса; 5) protected internal - доступ к члену возможен из одной сборки, либо из класса-наследника другой сборки.

Не указав модификатор доступа для члена, по умолчанию ему будет присвоен режим private.

Пример 1.1 Объявление простейшего класса

|  |
| --- |
| Задание. Объявите класс, который предназначен для операций с прямоугольникомРешение. namespace MyConsoleApplication{ // Объявление класса public class Rectangle { // Члены класса: // Поля. public int FSideA; public int FSideB; public string FName; // методы public int GetPerimeter () { return (FSideA + FSideB) \* 2; } public void SetName(string newName) { FName = newName; }  public string GetName() { return FName; } }} |

Чтобы от описания класса перейти к объекту, следует выполнить соответствующее объявление

Создание обьекта осуществляется по мере востребованности работы с ним следующим образом:

имя\_класса имя\_обьекта = **new** имя\_класса();

Пример 1.2 Создание объектов и работа с ними

|  |
| --- |
| Задание. Создайте 2 объекта класса Rectangle: первый “Большой прямоугольник” со сторонами 5 и 10 см; второй “Мальнький прямоугольник” со сторонами 2 и 3 см. Определите периметры фигур и выведите названия и периметры фигур на экран. Решение (для одной фигуры) class Program { static void Main(string[] args) {  // Создание первого объекта на основе класса Rectangle Rectangle myRectangle1 = new Rectangle(); // Установим значение полей myRectangle1.FSideA = 5; myRectangle1.FSideB = 10; // Вызовим метод  // изменения названия фигуры myRectangle1.SetName("Большой прямоугольник"); // определения периметра фигуры int Perimeter1 = myRectangle1.GetPerimeter();  // Выведим  // имя фигуры Console.WriteLine(myRectangle1.FName); // значение периметра Console.WriteLine(Perimeter1); Console.ReadKey(); // Создание второго объекта на основе класса Rectangle Rectangle myRectangle2 = new Rectangle(); // Установим значение полей myRectangle2.FSideA = 2; myRectangle2.FSideB = 3; // Вызовим метод  // изменения названия фигуры myRectangle2.SetName("Большой прямоугольник"); // определения периметра фигуры int Perimeter2 = myRectangle1.GetPerimeter();  // Выведим  // имя фигуры Console.WriteLine(myRectangle2.FName); // значение периметра Console.WriteLine(Perimeter2); Console.ReadKey(); }} |

Задание для самостоятельной работы

1. Объявите класс, который предназначен для ведения бухгалтерских записей на синтетическом счете;
2. Для объявленного класса определите методы;
3. Создайте несколько экземпляров класса и продемонстрируйте работу с ним: открытие счета, запись операций на счете, определение оборотов и конечного сальдо.

Контрольные вопросы

1) Что понимается под термином «класс»?

2) Какие элементы определяются в составе класса?

3) Каково соотношение понятий «класс» и «объект»?

4) Что понимается под термином «члены класса»?

5) Какие члены класса Вам известны?

6) Какие члены класса содержат код?

7) Какие члены класса содержат данные?

8) Перечислите пять разновидностей членов класса специфичных для

языка C#.

**Полный код.**

|  |
| --- |
| using System;using System.Collections.Generic;using System.Linq;using System.Text;using System.Threading.Tasks;namespace MyConsoleApplication{ // Объявление класса public class Rectangle { // Члены класса: // Поля. public int FSideA; public int FSideB; public string FName; // Методы // Расчет периметра public int GetPerimeter () { return (FSideA + FSideB) \* 2; } // Запись значения имени фигуры  public void SetName(string newName) { FName = newName; } // Чтение значения имени фигуры public string GetName() { return FName; } } class Program { static void Main(string[] args) {  Console.WriteLine("Работа с первым объектом"); // Создание первого объекта на основе класса Rectangle Rectangle myRectangle1 = new Rectangle(); // Установим значение полей myRectangle1.FSideA = 5; myRectangle1.FSideB = 10; // Вызовим метод  // изменения названия фигуры myRectangle1.SetName("Большой прямоугольник"); // определения периметра фигуры int Perimeter1 = myRectangle1.GetPerimeter();  // Выведим  // имя фигуры Console.WriteLine(myRectangle1.FName); // значение периметра Console.WriteLine(Perimeter1); Console.ReadKey(); Console.WriteLine("Работа со вторым объектом"); // Создание второго объекта на основе класса Rectangle Rectangle myRectangle2 = new Rectangle(); // Установим значение полей myRectangle2.FSideA = 2; myRectangle2.FSideB = 3; // Вызовим метод  // изменения названия фигуры myRectangle2.SetName("Маленький прямоугольник"); // определения периметра фигуры int Perimeter2 = myRectangle2.GetPerimeter(); // Выведим  // имя фигуры Console.WriteLine(myRectangle2.FName); // значение периметра Console.WriteLine(Perimeter2); Console.ReadKey(); } } } |

**Лабораторная работа №2. Классы и объекты: конструкторы и деструкторы**

**Цель работы:**

Познакомиться с основой объектного подхода в языке C#, изпользованием конструктуров и деструкторов.

**Теория**

Каждый раз, когда создается экземпляр класса вызывается специальный метод – конструктор. Конструкторы позволяют программисту задавать значения по умолчанию, ограничивать число установок и писать код, который является гибким и удобным для чтения.

Класс может иметь несколько конструкторов, принимающих различные аргументы. Конструкторы имеют то же имя, что и класс. Конструкторы вызывают с помощью оператора **new**. Конструктор не возвращает значение, даже типа void.

Конструктор без параметров называется конструктором по умолчанию. Если конструктор в классе явным образом не задан, компилятор C# для обеспечения создания экземпляров класса, предоставляет открытый конструктор по умолчанию (в этом случае все поля экземпляра класса ининцилизируются значениями по умолчанию принятыми для переменных).

Конструктор может использовать ключевое слово **base** для вызова конструктора базового класса.

Конструкторы могут быть отмечены модификаторами public (открытый), private (закрытый), protected (защищенный), internal (внутренний) или protectedinternal (защищенный внутренний). Эти модификаторы доступа определяют порядок доступа пользователей класса к конструкторам класса.

Пример 2.1 Работа с конструкторами

|  |
| --- |
| Задание. Объявите класс, предназначенный для операций с прямоугольником.В классе необходимо предусматреть два конструктора: конструктор по умолчанию (необходимо использовать для создания первого объекта «Прямоугольник 1х1») и конструктор, принимающий аргументы (необходимо использовать для создания второго объекта «Большой прямоугольник 5х10»). Создайте 2 объекта класса Rectangle: первый “Большой прямоугольник” со сторонами 5 и 10 см; второй “Прямоугольник” со сторонами 1 и 1 см. Определите периметры фигур и выведите названия и периметры фигур на экран..Решение. using System;using System.Collections.Generic;using System.Linq;using System.Text;using System.Threading.Tasks;namespace MyConsoleApplication{ // Объявление класса public class Rectangle { // Члены класса: // Поля. public int FSideA; public int FSideB; public string FName; // Конструкторы // Конструктор по умолчанию public Rectangle() { FSideA = 1; FSideB = 1; FName = "Прямоугольник 1х1"; } // Конструктор с аргументами public Rectangle(string Name, int SideA, int SideB) { FSideA = SideA; FSideB = SideB; FName = Name; } // Методы // Расчет периметра public int GetPerimeter () { return (FSideA + FSideB) \* 2; } // Чтение значения имени фигуры public void SetName(string newName) { FName = newName; } // Чтение значения имени фигуры public string GetName() { return FName; } } class Program { static void Main(string[] args) { Console.WriteLine("Работа с первым объектом"); // Создание первого объекта на основе класса Rectangle (Большой прямоугольник со сторонами 5 и 10) Rectangle myRectangle1 = new Rectangle("Большой прямоугольник",5,10); // Вызовим метод  // определения периметра фигуры int Perimeter1 = myRectangle1.GetPerimeter();  // Выведим  // имя фигуры Console.WriteLine(myRectangle1.FName); // значение периметра Console.WriteLine(Perimeter1); Console.ReadKey(); // Создание второго объекта на основе класса Rectangle (Прямоугольник со сторонами 1 и 1) Rectangle myRectangle2 = new Rectangle(); // Вызовим метод  // определения периметра фигуры int Perimeter2 = myRectangle2.GetPerimeter(); // Выведим  // имя фигуры Console.WriteLine(myRectangle2.FName); // значение периметра Console.WriteLine(Perimeter2); Console.ReadKey(); } } } |

Деструктор – метод, используемый для уничтожения экземпляров классов.

Класс может иметь только один деструктор. Деструкторы не могут наследоваться или перегружаться. Деструкторы невозможно вызвать: они запускаются автоматически (момент вызова определяется сборщиком мусора) или вызываются при выходе из программы. Деструктор не принимает модификаторы и не имеет параметров. Конструкторы имеют то же имя, что и класс, но с добавление слева имени символа «тильда» («~»).

В примере 2.2 используется ключевое слово **this**. Это указатель на объект, для которого был вызван метод. Ключевое слово this обеспечивает доступ к текущему экземпляру класса.

Одно из возможных применений ключевого слова this состоит в том, чтобы разрешать неоднозначность контекста, которая может возникнуть, когда входящий параметр назван так же, как поле данных данного типа.

Пример 2.2 Работа деструктора

|  |
| --- |
| Задание. Дополните класс Rectangle деструктором. Продемострируйте его работу. Решение. … // Конструкторы // Конструктор по умолчанию… // Конструктор с аргументами… // Деструктор ~Rectangle() { System.Diagnostics.Trace.WriteLine(this.GetName() + " уничтожен"); } // методы… |

Задание для самостоятельной работы

1. Используя решение задания, приведенного в лабораторной работе №1, дополните класс конструктором по умолчанию и двумя конструкторами, принимающим аргументы.

2. Используя решения задания, приведенного в лабораторной работе №1, дополните класс десконструктором.

3. Создайте несколько экземпляров класса и продемонстрируйте работу конструкторов (в частности при открытии активного и пассивного счета) и деструктора.

Контрольные вопросы

1. Для чего используется конструктор?
2. Что такое конструктор по умолчанию?
3. Для чего используется деструктор?
4. Когда запускается деструктор?

**Лабораторная работа №3. Классы и объекты: свойства. Инкапсуляция.**

**Цель работы:**

Познакомиться с основой объектного подхода в языке C#, использованием свойств. Познакомиться с реализацией принципа инкапсуляции на языке С#.

**Теория**

Инкапсуляция – один из основных принципов объектно-ориентированного программирования.

Согласно принципу инкапсуляции класс может задать уровень доступности каждого из членов по отношению к коду вне класса. Методы и поля, которые не предназначены для использования вне класса или сборки, могут быть скрыты, с целью ограничения потенциальной угрозы возникновения ошибок программного кода или его вредоносное использование.

Реализация принципа инкапсуляции обеспечивается за счет: 1) использования спецификаторов доступа для каждого члена класса; 2) использования свойств.

Спецификаторы доступа для членов класса были рассмотрены в лабораторной работе №1.

Свойства служат для организации упорядоченного доступа к полям класса. Как правило, свойство связано с закрытым полем класса и определяет методы его получения и установки.

Синтаксис свойства:

<уровень доступа> <тип><имя свойства > { get { код\_доступа} set { код\_доступа} }

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором (уровнем доступа) public), поскольку они входят в интерфейс объекта.

Код доступа представляет собой блоки операторов, которые выполняются при получении (get) или установке (set) свойства. Может отсутствовать либо часть get, либо set, но не обе одновременно.

Если отсутствует часть set, свойство доступно только для чтения (read-only), если отсутствует часть get, свойство доступно только для записи (write-only).

Метод записи обычно содержит действия по проверке допустимости устанавливаемого значения, метод чтения может содержать, например, поддержку счетчика обращений к полю.

В программе свойство выглядит как поле класса. При обращении к свойству автоматически вызываются указанные в нем методы чтения и установки.

Синтаксически чтение и запись свойства выглядят почти как методы. Метод get должен содержать оператор return, возвращающий выражение, для типа которого должно существовать неявное преобразование к типу свойства. В методе set используется параметр со стандартным именем value, который содержит устанавливаемое значение.

Вообще говоря, свойство может и не связываться с полем. Фактически, оно описывает один или два метода, которые осуществляют некоторые действия над данными того же типа, что и свойство. В отличие от открытых полей, свойства обеспечивают разделение между внутренним состоянием объекта и его интерфейсом и, таким образом, упрощают внесение изменений в класс.

Таким образом, свойство — это член, предоставляющий гибкий механизм для чтения, записи или вычисления значения частного (private) поля.

Пример 3.1 Работа со свойствами

|  |
| --- |
| Задание. Создайте объект класса Rectangle: “Большой прямоугольник” со сторонами 5 и 10 см; объявите все поля класса как закрытые; организуйте доступ к полям через свойства: Периметр (только для чтения) и Имя (для чтения и записи) .Решение. using System;using System.Collections.Generic;using System.Linq;using System.Text;using System.Threading.Tasks;namespace MyConsoleApplication{ // Объявление класса public class Rectangle { // Члены класса: // Поля. private int FSideA; private int FSideB; private string FName; // Конструктор с аргументами public Rectangle(string Name, int SideA, int SideB) { FSideA = SideA; FSideB = SideB; FName = Name; }  // Свойства // Только для чтения public int Perimeter  { get {return (FSideA + FSideB) \* 2;} } // Для чтения и для записи public string Name { set { FName=value; } get { return FName; } }  } class Program { static void Main(string[] args) {  // Создание объекта на основе класса Rectangle (Большой прямоугольник со сторонами 5 и 10) Rectangle myRectangle = new Rectangle("Большой прямоугольник",5,10); // Выведим свойства фигуры // имя  Console.WriteLine(myRectangle.Name); // значение периметра Console.WriteLine(myRectangle.Perimeter); Console.ReadKey();  } } } |

Таким образом, согласно принципу инкапсуляции любой класс рассматривается как черный ящик (пользователь класса может видеть и использовать только тот набор декларируемых свойств и методов класса (так называемую интерфейсную часть), который разрешил ему разработчик класса). При этом внутренняя реализация класса пользователю недоступна.

Задание для самостоятельной работы

1. Используя решения задания, приведенного в лабораторной работе №1 и №2, дополните класс свойствами разных видов (только для чтения, только для записи, для записи и для чтения).

2. Создайте несколько экземпляров класса и продемонстрируйте работу индексаторами

Контрольные вопросы

1. Что такое инкапсуляция?
2. Для чего используется свойства?
3. Как вызываются методы чтения и установки свойства?
4. Каково содержание метода чтения свойства?
5. Каково содержание метода установки свойства?

**Лабораторная работа №4. Классы и объекты: индексаторы**

**Цель работы:**

Познакомиться с основой объектного подхода в языке C#, использованием индексаторов.

**Теория**

Индексатор представляет собой разновидность свойства. Если у класса есть скрытое поле, представляющее собой массив, то с помощью индексатора можно обратиться к элементу этого массива, используя имя объекта и номер элемента массива в квадратных скобках.

Синтаксис индексатора аналогичен синтаксису свойства:

<уровень доступа> <тип>**this** [список\_параметров] { get { код\_доступа} set { код\_доступа} }

Индексаторы чаще всего объявляются спецификатором public, поскольку они входят в интерфейс объекта.

Код доступа представляет собой блоки операторов, которые выполняются при получении ( get ) или установке значения ( set ) элемента массива. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, индексатор доступен только для чтения (read-only), если отсутствует часть get, индексатор доступен только для записи (write-only).

Синтаксически чтение и запись индексатора выглядят почти как методы. Метод get должен содержать оператор return, возвращающий выражение, для типа которого должно существовать неявное преобразование к типу свойства. В методе set используется параметр со стандартным именем value, который содержит устанавливаемое значение.

Список параметров содержит одно или несколько описаний индексов, по которым выполняется доступ к элементу. Чаще всего используется один индекс целого типа. Однско, C# не ограничивает тип индексатора типом «integer». Например, может оказаться полезным использование в индексаторе строки. Такой индексатор можно реализовать, выполнив поиск строки в коллекции и возвратив соответствующее значением. Методы доступа можно перегружать, версии типа «string» и «integer» могут сосуществовать

Индексаторы могут иметь более одного формального параметра, например, при доступе к двухмерному массиву

Пример 4.1 Работа с индексатором

|  |
| --- |
| Задание. Создайте объект класса Rectangle: “Большой прямоугольник” со сторонами 5 и 10 см; объявите поле «Имя» данного класса, а также поле «Размеры сторон», содержащее информацию о размерах всех сторонах прямоугольника. Организуйте доступ к полю «Имя» посредством свойства, а к полю «Размеры сторон» посредством индексатора.Решение. using System;using System.Collections.Generic;using System.Linq;using System.Text;using System.Threading.Tasks;namespace MyConsoleApplication{ // Объявление класса public class Rectangle { // Члены класса: // Поля. private string FName; private int[] FSide = new int[2]; // декларируем массив с двумя элементами // Индексатор с целочисленным парметром (индексом) public int this[int index] { get { return FSide[index]; } set { FSide[index] = value; } } // Конструктор с аргументами public Rectangle(string Name, int SideA, int SideB) { FSide[0] = SideA; FSide[1] = SideB; FName = Name; }  // Свойства // Только для чтения public int Perimeter { get {return (FSide[0] + FSide[1]) \* 2; } } // Для чтения и для записи public string Name { set { FName=value; } get { return FName; } }  } class Program { static void Main(string[] args) {  // Создание объекта на основе класса Rectangle (Большой прямоугольник со сторонами 5 и 10) Rectangle myRectangle = new Rectangle("Большой прямоугольник",5,10); // Выведим свойства фигуры // имя  Console.WriteLine(myRectangle.Name); // значение сторон (до изменения) Console.WriteLine(myRectangle[0]); Console.WriteLine(myRectangle[1]); Console.ReadKey();  // Переустановим значение длин сторон прямоугольника myRectangle[0] = 10; myRectangle[1] = 20; // Выведим свойства фигуры // значение сторон (после изменения) Console.WriteLine(myRectangle[0]); Console.WriteLine(myRectangle[1]); Console.ReadKey(); // значение периметра Console.WriteLine(myRectangle.Perimeter); Console.ReadKey();  } } } |

Задание для самостоятельной работы

1. Используя решение задания, приведенного в лабораторной работе №1,2,3, дополните класс индексатором

2. Создайте несколько экземпляров класса и продемонстрируйте работу индексатора

Контрольные вопросы

1. В каких случаях применение индексатора является целесообразным?
2. Каким образом осуществляется чтение и запись индексатора?

**Лабораторная работа №5. Классы и объекты: наследование**

**Цель работы:**

Познакомиться с реализацией принципа наследования на языке С#.

**Теория**

Наследование является одной из основных характеристик (или базовых понятий) объектно-ориентированного программирования.

**Наследование** — это механизм, который дает возможность создавать новый (производный) класс на основе определения уже существующего (базового) класса.

 С помощью механизма наследования производный класс наследует все свойства и поведение (все методы и свойства интерфейса базового класса автоматически появляются в интерфейсе производного класса), представленные в базовом классе, при этом позволяя их расширять и изменять.

Производный класс может иметь только один непосредственный базовый класс. Однако наследование является транзитивным. Если ClassC является производным от ClassB, и ClassB является производным от ClassA, ClassC наследует члены, объявленные в ClassB и ClassA.

Наследование применяется для следующих взаимосвязанных целей: 1) исключения из программы повторяющихся фрагментов кода; 2) упрощения модификации программы; 3) упрощения создания новых программ на основе существующих.

При описании класса имя его базового класса записывается в заголовке класса после двоеточия (если имя предка не указано, предком считается базовый класс всей иерархии System.Object):

[уровень доступа] **class** <имя производного класса>:<имя базового класса >

{

// Описание членов класса

}

**Поля, методы и свойства класса наследуются**, поэтому при желании заменить элемент базового класса новым элементом следует явным образом указать компилятору свое намерение с помощью ключевого слова new. Соответственно синтаксис для полей, свойств и методов выглядит следующим образом:

**new** <уровень доступа> <тип переменной> <имя переменной>;

**new** <уровень доступа> <тип><имя свойства > { get { код\_доступа} set { код\_доступа} }

**new** <уровень доступа> <возвращаемый тип><имя метода > (<список параметров>){тело\_метода}

Из производного класса можно получить доступ к открытым (public), защищенным (protected), внутренним (internal) и защищенным внутренним (protected internal) членам базового класса.

Хотя производный класс и наследует закрытые члены базового класса, он не может получить доступ к этим членам. Однако все эти закрытые члены все же присутствуют в производном классе и могут выполнять ту же работу, что и в самом базовом классе (например, если защищенный метод базового класса имеет доступ к закрытому полю, то это поле должно присутствовать в производном классе для правильной работы унаследованного метода базового класса).

**Конструкторы не наследуются**, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными далее правилами:

1. Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров.
2. Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса. Таким образом, каждый конструктор инициализирует свою часть объекта.
3. Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации. Вызов выполняется с помощью ключевого слова base. Вызывается та версия конструктора, список параметров которой соответствует списку аргументов, указанных после слова base.

Пример 5.1 Наследование ()

|  |
| --- |
| Задание. Создайте два класса:«Shape» (базовый класс), предназначенный для создания на своей основе классов любых геометрических фигур, и обеспечивающий возможность чтения и записи их наименования. «Rectangle» (производный класс), предназначенный для описания состояния и поведения прямоугольников как геометрических фигур.Решение. using System;using System.Collections.Generic;using System.Linq;using System.Text;using System.Threading.Tasks;namespace MyConsoleApplication{ // Объявление класса <Фигура> public class Shape { // Члены класса: // Поля. protected string FName; protected int FSideA; protected int FSideB;  // Конструкторы public Shape(string Name, int SideA, int SideB) { FSideA = SideA; FSideB = SideB; this.Name = Name; } // Свойства public string Name //Наименование фигуры { set { FName = value + " (неопределенная фигура)"; } get { return FName; } } } public class Rectangle : Shape { // Члены класса: // Конструкторы public Rectangle(string Name, int SideA, int SideB): base(Name,SideA,SideB){} // Свойства public int Perimeter // свойство, дополняющее элементы базового класса  { get { return (FSideA + FSideB) \* 2; } } // Свойства new public string Name // свойство, заменяющее элемент базового класса  { set { FName = value + " (прямоугольник)"; } get { return FName; } } } class Program { static void Main(string[] args) { // Создание объекта на основе класса Rectangle (Большой прямоугольник со сторонами 5 и 10) Rectangle myRectangle = new Rectangle("Большой прямоугольник", 5, 10); // Выведем свойства фигуры // имя  Console.WriteLine(myRectangle.Name); // значение периметра Console.WriteLine(myRectangle.Perimeter); Console.ReadKey(); } }} |

Задание для самостоятельной работы

1. Используя решение заданий, приведенных в лабораторных работах №1,2,3,4, создайте один базовый класс и несколько (не менее двух) производных классов
2. Создайте экземпляры производных классов и продемонстрируйте их работу

Контрольные вопросы

1. Что такое наследование?
2. Для каких целей применяют наследование?
3. Какие члены класса наследуются?
4. Какие члены класса не наследуются?
5. Каков порядок вызова конструкторов при наследовании?

**Лабораторная работа №6. Классы и объекты: виртуальные методы и свойства, полиморфизм и абстрактные классы**

**Цель работы:**

Познакомиться с использованием виртуальных методов и свойств, абстрактных и бесплодных классов. Познакомиться с реализацией принципа полиморфизма на языке С#.

**Теория**

Во время разработки программы удобно оперировать объектами одной иерархии единообразно, то есть использовать один и тот же программный код для работы с экземплярами разных классов. Желательно иметь возможность описать: 1) объект, в который во время выполнения программы заносятся ссылки на объекты разных классов иерархии; 2) контейнер, в котором хранятся объекты разных классов, относящиеся к одной иерархии; 3) метод, в который могут передаваться объекты разных классов иерархии; 4) метод, из которого в зависимости от типа вызвавшего его объекта вызываются соответствующие методы.

Все это возможно благодаря тому, что объекту базового класса в С# можно присвоить объект производного класса, однако при этом для него вызываются только методы и свойства, определенные в базовом классе (см. пример 6.1).

Пример 6.1 Присвоение объекту базового класса объекта производного класса (пример несоответствия вызываемых методов типу объекта)

|  |
| --- |
| using System;using System.Collections.Generic;using System.Linq;using System.Text;using System.Threading.Tasks;namespace MyConsoleApplication{ // Объявление класса <Фигура> public class Shape { // Члены класса: // Поля. protected string FName; public Shape() { FName = "Неопределенная фигура"; } public string Name //Наименование фигуры { get { return FName + " (неопределенная фигура)"; } } } public class Rectangle : Shape { // Члены класса: // Конструкторы public Rectangle() { FName = "Прямоугольник"; } new public string Name // свойство, заменяющее элемент базового класса  { get { return FName; } }} public class Circle : Shape { // Члены класса: // Конструкторы public Circle() { FName = "Окружность"; } new public string Name // свойство, заменяющее элемент базового класса  { get { return FName; } } } class Program { static void Main(string[] args) { // Создание объектов базового класса  // на основе производных класcов  Shape myShape1 = new Rectangle(); Shape myShape2 = new Circle(); // на основе базового класса Shape myShape3 = new Shape(); // Выведим свойства фигуры // имя  Console.WriteLine(myShape1.Name); Console.WriteLine(myShape2.Name); Console.WriteLine(myShape3.Name); Console.ReadKey(); Console.WriteLine(""); // То же самое, но с использование массива Shape[] MyShape = new Shape[3]; MyShape[0] = new Rectangle(); MyShape[1] = new Circle(); MyShape[2] = new Shape(); foreach (Shape elem in MyShape) Console.WriteLine(elem.Name); Console.ReadKey(); } }} |

Таким образом, возможность доступа к элементам класса определяется типом ссылки, а не типом объекта, на который она указывает.

Причина этого состоит в том, что компилятор еще до выполнения программы должен решить, какой метод вызывать, и вставить в код фрагмент, передающий управление на этот метод (этот процесс называется ранним связыванием). При этом компилятор может руководствоваться только типом переменной, для которой вызывается метод или свойство. То, что в этой переменной в разные моменты времени могут находиться ссылки на объекты разных типов, компилятор учесть не может.

Следовательно, если мы хотим, чтобы вызываемые методы соответствовали типу объекта, необходимо отложить процесс связывания до этапа выполнения программы, а точнее — до момента вызова метода, когда уже точно известно, на объект какого типа указывает ссылка. Такой механизм в С# есть — он называется поздним связыванием и реализуется с помощью так называемых виртуальных методов.

Объявление метода виртуальным означает, что все ссылки на этот метод будут разрешаться по факту его вызова, то есть не на стадии компиляции, а во время выполнения программы.

Для обозначения виртуального метода и виртуального используется ключевое слово **virtual**, которое записывается в заголовке метода или свойства базового класса следующим образом:

**virtual** <уровень доступа> <тип><имя свойства > { get { код\_доступа} set { код\_доступа} }

**virtual** <уровень доступа> <возвращаемый тип><имя метода > (<список параметров>){тело\_метода}

Если в производном классе требуется переопределить виртуальный метод или виртуальное свойство, используется ключевое слово **override**:

**override** <уровень доступа> <тип><имя свойства > { get { код\_доступа} set { код\_доступа} }

**override** <уровень доступа> <возвращаемый тип><имя метода > (<список параметров>){тело\_метода}

Пример 6.2 Использование виртуальных методов (свойств)

|  |
| --- |
| Задание. Изменить программный код, приведенный в пример 6.1, таким образом, что бы вызываемые методы соответствовали типу объектаРешение. using System;using System.Collections.Generic;using System.Linq;using System.Text;using System.Threading.Tasks;namespace MyConsoleApplication{ // Объявление класса <Фигура> public class Shape { // Члены класса: // Поля. protected string FName; public Shape() { FName = "Неопределенная фигура"; } virtual public string Name //Наименование фигуры { get { return FName + " (неопределенная фигура)"; } } } public class Rectangle : Shape { // Члены класса: // Конструкторы public Rectangle() { FName = "Прямоугольник"; } override public string Name // свойство, заменяющее элемент базового класса  { get { return FName; } }} public class Circle : Shape { // Члены класса: // Конструкторы public Circle() { FName = "Окружность"; } override public string Name // свойство, заменяющее элемент базового класса  { get { return FName + ""; } } } class Program { static void Main(string[] args) { // То же самое, но с использование массива Shape[] MyShape = new Shape[3]; MyShape[0] = new Rectangle(); MyShape[1] = new Circle(); MyShape[2] = new Shape(); foreach (Shape elem in MyShape) Console.WriteLine(elem.Name); Console.ReadKey(); } }} |

Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса. Это требование вполне естественно, если учесть, что одноименные методы, относящиеся к разным классам, могут вызываться из одной и той же точки программы.

Виртуальные методы базового класса определяют интерфейс всей иерархии. Этот интерфейс может расширяться в потомках за счет добавления новых виртуальных методов. Переопределять виртуальный метод в каждом из потомков не обязательно: если он выполняет устраивающие потомка действия, метод наследуется.

С помощью виртуальных методов реализуется один из основных принципов объектно-ориентированного программирования — полиморфизм. Это слово в переводе с греческого означает «много форм», что в данном случае означает «один вызов — много методов». Применение виртуальных методов обеспечивает гибкость и возможность расширения функциональности класса (см. пример 6.3).

Пример 6.3 Полиморфизм

|  |
| --- |
| Задание. Создайте три класса: «Shape» (базовый класс), предназначенный для создания на своей основе классов любых геометрических фигур, и обеспечивающий возможность чтения и записи их наименования. «Rectangle» (производный класс), предназначенный для описания состояния и поведения прямоугольников как геометрических фигур.«Circle» (производный класс), предназначенный для описания состояния и поведения окружностей как геометрических фигур.Создайте массив объектов типа «Shape» и продемонстрируйте реализацию принципа полиморфизма при вычислении периметров фигур.Решение. using System;using System.Collections.Generic;using System.Linq;using System.Text;using System.Threading.Tasks;namespace MyConsoleApplication{ // <Фигура>  public class Shape { protected string FName; protected double FSideA,FSideB; public Shape() {FName = "Неопределенная фигура";} public Shape(string Name, double SideA, double SideB){FName = Name; FSideA = SideA; FSideB = SideB;} virtual public double GetPerimeter() { return (FSideA + FSideB) \* 2; } virtual public string GetName() {return FName;} } // <Прямоугольник>  public class Rectangle : Shape { public Rectangle(string Name, double SideA, double SideB) : base(Name, SideA, SideB) {} } // <Окружность>  public class Circle : Shape { public Circle(string Name, double Radius): base(Name, Radius, 0){} override public double GetPerimeter(){return 2 \* Math.PI \* FSideA;} }class Program { static void Main(string[] args) { Shape[] MyShape = new Shape[2]; MyShape[0] = new Rectangle("Прямоугольник", 5,10); MyShape[1] = new Circle("Окружность", 5); foreach (Shape elem in MyShape) Console.WriteLine(elem.GetName()+" - "+elem.GetPerimeter()); Console.ReadKey(); } }} |

При описании классов рекомендуется определять в качестве виртуальных те методы, которые в производных классах должны реализовываться по-другому.

Если во всех классах иерархии метод будет выполняться одинаково, его лучше определить как обычный метод.

Таким образом, при наследовании производный класс получает от базового класса все методы, поля, свойства и события базового класса. При разработке производного класса можно поступить следующим образом: 1) либо наследовать члены ближайшего базового класса без переопределения; 2) либо заменить реализацию отдельных членов производного класса, скрывая при этом их реализацию в базовом классе; 3) либо переопределить виртуальные члены в базовом классе.

При создании иерархии объектов для исключения повторяющегося кода часто бывает логично выделить их общие свойства в один родительский класс. При этом может оказаться, что создавать экземпляры такого класса не имеет смысла, потому что никакие реальные объекты им не соответствуют. Такие классы называют абстрактными.

Абстрактный класс объявляется со спецификатором **abstract.** Абстрактный класс служит только для порождения потомков. Как правило, в нем задается набор методов, которые каждый из потомков будет реализовывать по-своему. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах.

Абстрактный класс задает интерфейс для всей иерархии, при этом методам класса может не соответствовать никаких конкретных действий. В этом случае методы имеют пустое тело и объявляются со спецификатором abstract. В производном классе методы переопределяются с помощью спецификатора overrite.

В некоторых случаях возникает необходимость запретить наследовать от определенного класса. Ключевое слово **sealed** позволяет описать такой класс, который принято называть бесплодным.

Задание для самостоятельной работы

1. Используя решение заданий, приведенных в лабораторных работах №1,2,3,4,6, создайте один базовый класс и несколько (не менее двух) производных классов. Создайте несколько виртуальных свойств и методов, в том числе используя принцип полиморфизма.
2. Создайте экземпляры базового и производных классов, продемонстрируйте их работу, в том числе при присвоение объекту базового класса объекта производного класса

Контрольные вопросы

1. Что такое полиморфизм?
2. Для каких целей применяют полиморфизм?
3. Для чего предназначены виртуальные методы?
4. Для чего предназначены абстрактные классы?
5. Что представляет собой бесплодный класс?
6. В чем отличие механизма быстрого связывания от механизма позднего связывания?

**Задания для самостоятельной работы**

Для каждого варианта дана предметная область. Основные сущности (объекты), которые характеризуются определенным состоянием и поведением.

**Вариант №1**

Вы работаете в страховой компании. Вашей задачей является отслеживание финансовой деятельности компании.

Компания имеет различные филиалы по всей стране. Каждый филиал характеризуется названием, адресом и телефоном. Деятельность компании организована следующим образом: к Вам обращаются различные лица с целью заключения договора о страховании. В зависимости от принимаемых на страхование объектов и страхуемых рисков, договор заключается по определенному виду страхования (например, страхование автотранспорта от угона, страхование домашнего имущества, добровольное медицинское страхование). При заключении договора Вы фиксируете дату заключения, страховую сумму, вид страхования, тарифную ставку и филиал, в котором заключался договор.

Договоры заключают страховые агенты. Помимо информации об агентах (фамилия, имя, отчество, адрес, телефон, код филиала), нужно еще хранить филиал, в котором работают агенты.

Кроме того, исходя из базы данных, нужно иметь возможность рассчитывать заработную плату агентам. Заработная плата составляет некоторый процент от страхового платежа (страховой платеж это страховая сумма, умноженная на тарифную ставку). Процент зависит от вида страхова-ния, по которому заключен договор.

Внести в структуру таблиц изменения, учитывающие эти факты, и изменить существующие запросы. Добавить новые запросы.

Размер страховой суммы расчитывается в зависимости от вида страхования и траифной ставки

Сущности (объекты)

1. Договоры (Номер договора, Дата заключения, Страховая сумма, Тарифная ставка, Код филиала, Код вида страхования).
2. Вид страхования (Код вида страхования, Наименование).
3. Филиал (Код филиала, Наименование филиала, Адрес, Телефон).

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MyConsoleApplication

{

 // Объявление класса <Фигура>

 public class Figure

 {

 // Члены класса:

 // Поля.

 protected string FName;

 // Свойства

 public string Name //Наименование фигуры

 {

 set { FName = value + " (неопределенная фигура)"; }

 get { return FName; }

 }

 }

 public class Rectangle : Figure

 {

 // Члены класса:

 // Поля.

 protected int FSideA;

 protected int FSideB;

 // Конструкторы

 public Rectangle()

 {

 FSideA = 0; FSideB = 0; FName = "";

 }

 public Rectangle(string Name, int SideA, int SideB)

 {

 FSideA = SideA; FSideB = SideB; this.Name = Name;

 }

 // Свойства

 public int Perimeter // свойство, дополняющее элементы базового класса

 {

 get { return (FSideA + FSideB) \* 2; }

 }

 new public string Name // свойство, заменяющее элемент базового класса

 {

 set { FName = value + " (прямоугольник)"; }

 get { return FName; }

 }

}

 class Program

 {

 static void Main(string[] args)

 {

 // Создание объекта на основе класса Rectangle (Большой прямоугольник со сторонами 5 и 10)

 Rectangle myRectangle = new Rectangle("Большой прямоугольник", 5, 10);

 // Выведим свойства фигуры

 // имя

 Console.WriteLine(myRectangle.Name);

 // значение периметра

 Console.WriteLine(myRectangle.Perimeter);

 Console.ReadKey();

 }

 }

}

1. Сборка (assembly) – это готовый функциональный модуль в виде exe либо dll файла (файлов), который содержит скомпилированный код для .NET. Сборка предоставляет возможность повторного использования кода. [↑](#footnote-ref-1)