Лабораторная работа № 3.
Разработка классов различной природы

Задание 1. Разработать базовый класс Фигура2D и создать производные классы Квадрат и Прямоугольник. Учесть все возможные действия, которые могут выполняться с данными фигурами. Предусмотреть возможность работы с различными типами данных. Продемонстрировать использование этих классов в консольном приложении.

Задание 2. Разработать интерфейс Фигура2D и создать классы Квадрат и Прямоугольник. Учесть все возможные действия, которые могут выполняться с данными фигурами. Продемонстрировать использование этих классов в консольном приложении.

Задание 3. Разработать базовый класс Фигура3D и создать производные классы Куб и Параллелепипед. Учесть все возможные действия, которые могут выполняться с данными фигурами. Предусмотреть возможность работы с различными типами данных. Продемонстрировать использование этих классов в консольном приложении.

Задание 4. Разработать интерфейс Треугольник и создать классы ПрямоугольныйТреугольник, РавностороннийТреугольник и РавнобедренныйТреугольник. Учесть все возможные действия, которые могут выполняться с данными фигурами. Продемонстрировать использование этих классов в консольном приложении.

Задание 5. Разработать класс Репка, используя для разработки одноименною сказку. Продемонстрировать работу данного класса. Результат вывода сказки должен удовлетворять правилам русского языка.

Задание 6. Разработать класс Репка, используя для разработки одноименную сказку. Предусмотреть возможность участия в сказке других героев (см. рис.1 и 2). Продемонстрировать работу данного класса. Результат вывода сказки должен удовлетворять правилам русского языка.

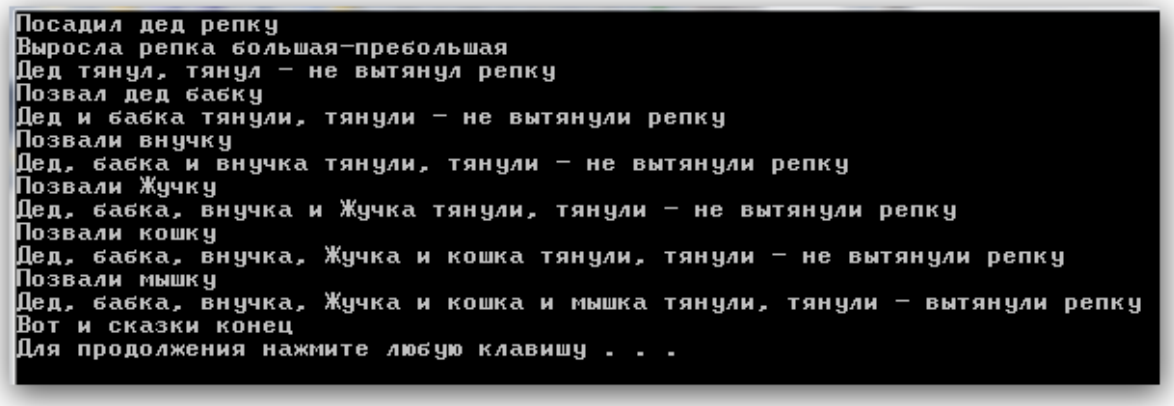

Рис. 1. Результат работы класса Репка, учитывающего возможность ввода любых данных.

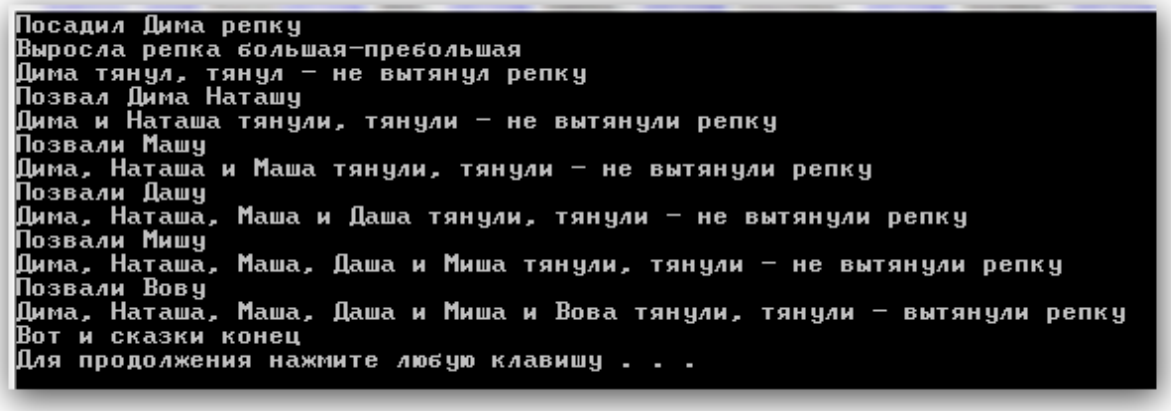

Рис. 2. Результат работы класса Репка, учитывающего возможность ввода любых данных.

Задание 7. Разработать базовый класс Студент и создать производные классы Бакалавр и Магистр. Учесть все возможные действия, которые могут выполняться с данными фигурами. Предусмотреть возможность работы с различными типами данных. Продемонстрировать использование этих классов в консольном приложении.

Задание 8. Разработать интерфейс Студент и создать классы Бакалавр и Магистр. Учесть все возможные действия, которые могут выполняться с данными фигурами. Продемонстрировать использование этих классов в консольном приложении.

Каждое задание выполняется в отдельном проекте. Имена проектов

Лаб_ФИО_3_номерзадания.

Наследование и производные классы

Функциональные возможности существующего класса можно расширить путем создания нового класса, производного от существующего. Производный класс наследует все свойства базового класса, и можно добавлять или переопределять методы и свойства в зависимости от необходимости.

В языке C# как наследование, так и реализация интерфейса определяются оператором :. Базовый класс должен всегда занимать крайнее левое положение в объявлении класса.

C# не поддерживает множественное наследование. Это значит, что классы не могут наследовать от нескольких классов. Однако для этой цели можно использовать интерфейсы.

В следующем коде определяется класс с именем CoOrds с двумя закрытыми переменными членов x и y, представляя положение точки. Эти переменные вызываются через свойства с именем X и Y соответственно.

```
public class CoOrds
{
    private int x, y;
    public CoOrds()  // Конструктор
    {
        x = 0;
        y = 0;
    }
    public int X
    {
        get { return x; }
        set { x = value; }
    }
    public int Y
    {
        get { return y; }
        set { y = value; }
    }
}
```

Производный от класса CoOrds новый класс с именем ColorCoOrds создается следующим образом.

```
public class ColorCoOrds : CoOrds
```

Затем, ColorCoOrds наследует все поля и методы базового класса, к которым можно добавить новые, чтобы получить дополнительные возможности в производном классе, если необходимо. В этом примере добавляется закрытый член и методы доступа, чтобы добавить цвет в класс.

```
public class ColorCoOrds : CoOrds
{
    private System.Drawing.Color screenColor;
    public ColorCoOrds()  // Конструктор
    {
        screenColor = System.Drawing.Color.Red;
    }
    public System.Drawing.Color ScreenColor
    {
        get { return screenColor; }
        set { screenColor = value; }
    }
}
```

Конструктор производного класса неявно вызывает конструктор для базового класса При наследовании все конструкторы базового класса вызываются до конструкторов производного класса для того, чтобы классы присутствовали в иерархии классов.

Для доступа к членам и методам производного класса нельзя использовать ссылку на базовый класс, даже если она может содержать допустимую ссылку на объект производного типа.

Ссылаться на производный класс можно при помощи неявной ссылки на производный тип.

```
ColorCoOrds color1 = new ColorCoOrds();
CoOrds coords1 = color1;
```

В этом коде ссылка на базовый класс coords1 содержит копию ссылки color1.

Ключевое слово base

Доступ к членам базового класса во вложенном классе можно получить даже если эти базовые члены переопределены в суперклассе при помощи ключевого слова base. Например, можно создать производный класс, содержащий метод с той же подписью, что и в базовом классе. Если перед методом поставить ключевое слово new, то это будет означать, что метод является абсолютно новым, принадлежащим производному классу. С помощью ключевого слова base можно по-прежнему создать метод для доступа к исходному методу в базовом классе.

Предположим, что базовый класс CoOrds имеет метод с именем Invert(), который меняет местами координаты x и y. Этот метод можно подменить в производном классе ColorCoOrds следующим кодом.

```
public new void Invert()
{
    int temp = X;
    X = Y;
    Y = temp;
    screenColor = System.Drawing.Color.Gray;
}
```

Этот метод меняет местами x и y и затем задает серый цвет координат. Доступ к базовой реализации этого метода можно предоставить путем создания другого метода ColorCoOrds, например следующего:

```
public void BaseInvert()
{
    base.Invert();
}
```

Затем можно вызвать базовый метод в объекте ColorCoOrds путем вызова метода BaseInvert().

```
ColorCoOrds color1 = new ColorCoOrds();
color1.BaseInvert();
```

Такого же результата можно добиться, назначив ссылку базовому классу на экземпляр ColorCoOrds и вызвав его методы:

```
CoOrds coords1 = color1;
coords1.Invert();
```

Выбор конструкторов

Построение объектов базового класса всегда выполняется до любого производного класса. Так, конструктор базового класса выполняется перед конструктором производного класса. Если базовый класс имеет несколько конструкторов, производный класс может выбрать вызываемый

конструктор. Например, можно изменить класс CoOrds для добавления второго конструктора, как показано ниже:

```
public class CoOrds
{
    private int x, y;
    public CoOrds()
    {
        x = 0;
        y = 0;
    }
    public CoOrds(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Потом можно изменить ColorCoOrds и использовать определенный класс доступных конструкторов при помощи ключевого слова base:

```
public class ColorCoOrds : CoOrds
{
    public System.Drawing.Color color;
    public ColorCoOrds() : base ()
    {
        color = System.Drawing.Color.Red;
    }
    public ColorCoOrds(int x, int y) : base (x, y)
    {
        color = System.Drawing.Color.Red;
    }
}
```

Переопределение методов

Производный класс может переопределить метод базового класса, предоставив новую реализацию для объявленного метода. Переопределение методов доступа к свойствам, а также методов выполняется практически одинаково.

Виртуальные методы

Метод базового класса, который должен быть переопределен в производном классе объявляется, при помощи модификатора **virtual**. В производном классе для объявления переопределенного метода используется модификатор **override**.

Модификатор override обозначает метод или свойство производного класса, который заменяет его тем же именем и подписью в базовом классе. Базовый метод, который будет перезаписан, должен быть объявлен как virtual, abstract или override: таким способом нельзя переопределить статический метод или метод, не являющийся виртуальным. Как переопределенный, так и переопределяемый метод или свойство должны иметь одинаковые модификаторы уровня доступа.

В следующем примере показан виртуальный метод с именем StepUp, переопределенный в производном классе модификатором переопределения:

```
public class CountClass
{
```

```
    public int count;
    public CountClass(int startValue)  // Конструктор
    {
        count = startValue;
    }
    public virtual int StepUp()
    {
        return ++count;
    }
}
class Count100Class : CountClass
{
    public Count100Class(int x) : base(x)  // Конструктор
    {
    }
    public override int StepUp()
    {
        return ((base.count) + 100);
    }
}
class TestCounters
{
    static void Main()
    {
        CountClass counter1 = new CountClass(1);
        CountClass counter100 = new Count100Class(1);
        System.Console.WriteLine("Вычисление в базовом классе = {0}",
counter1.StepUp());
        System.Console.WriteLine("Вычисление в производном классе =
{0}", counter100.StepUp());
    }
}
```

При выполнении этого кода можно заметить, что конструктор производного класса использует тело метода в базовом классе, позволяя инициализировать член счетчика не дублируя этот код. Ниже приведены выходные данные.

Вычисление в базовом классе = 2

Вычисление в производном классе = 101

Абстрактные классы

Абстрактный класс объявляет один или несколько методов или свойств в качестве абстрактных. Такие методы не имеют реализации в классе, объявляющем их, однако абстрактный класс может также содержать неабстрактные методы, то есть методы для которых предоставляется реализация. Экземпляр абстрактного класса не может быть создан непосредственно – только как производный класс. Такие производные классы должны предоставлять реализации для всех абстрактных методов и свойств при помощи ключевого слова override, если только сам производный член не объявлен абстрактным.

В следующем примере объявляется абстрактный класс Employee (Сотрудник). Также, можно создать производный класс с именем Manager (Менеджер), предоставляющий реализацию абстрактного метода Show(), определенного в классе Employee.

```
public abstract class Employee
```

```
{
    protected string name;
    public Employee(string name)  // Конструктор
    {
        this.name = name;
    }
    public abstract void Show();  // Абстрактный метод Show
}
public class Manager: Employee
{
    public Manager(string name) : base(name) {}  // Конструктор
    public override void Show()  //Переопределение абстрактного метода
Show
    {
        System.Console.WriteLine("Имя : " + name);
    }
}
class TestEmployeeAndManager
{
    static void Main()
    {
        // Создание экземпляра класса Manager и связывание его
        // со ссылкой Manager
        Manager m1 = new Manager("Вася Пупкин");
        m1.Show();
        // Создание экземпляра класса Manager и связывание его
        // со ссылкой Employee:
        Employee ee1 = new Manager("Ваня Иванов");
        ee1.Show();  //Вызов метода Show класса Manager
    }
}
```

Этот код вызывает реализацию Show(), предоставляемую классом Manager и выводит имена сотрудников на экран. Ниже приведены выходные данные.

Интерфейсы

Интерфейс – это своего рода скелет класса, содержащий подписи методов, но не включающий реализации методов. В этом отношении интерфейсы напоминают абстрактные классы, содержащие только абстрактные методы.

Все члены интерфейса являются открытыми по определению, и интерфейс не может содержать констант, полей (закрытых элементов данных), конструкторов, деструкторов или какого-либо типа статического члена. Если для членов интерфейса указать любой модификатор, компилятор создаст ошибку.

Для реализации такого интерфейса можно создать производные от интерфейса классы. Такие производные классы должны предоставлять реализации для всех методов интерфейса, если только сам производный класс не объявлен абстрактным.

В определении интерфейса свойство указывает только его тип, а также определяет, доступно ли он только для чтения, только для записи или для чтения/записи при помощи только ключевых слов get и set. В представленном ниже интерфейсе объявлено одно свойство только для чтения.

```
public interface ICDPlayer
{
```

```
    void Play();  // сигнатура метода
    void Stop();  // сигнатура метода
    int FastForward(float numberOfSeconds);
    int CurrentTrack  // свойство read-only
    {
        get;
    }
}
```

Класс может наследовать от этого интерфейса. Реализующий класс должен предоставлять определения для всех методов, а также любые обязательные методы доступа к свойствам, как показано ниже:

```
public class CDPlayer : ICDPlayer
{
    private int currentTrack = 0;
    // Реализация методов, определенных в интерфейсе
    public void Play()
    {
        // Код для запуска CD...
    }
    public void Stop()
    {
        // Код, чтобы остановить CD...
    }
    public int FastForward(float numberOfSeconds)
    {
        // Код для перемотки вперед CD с помощью numberOfSeconds...
        return 0;  //возврат кода успеха
    }
    public int CurrentTrack  // свойство read-only
    {
        get
        {
            return currentTrack;
        }
    }
    // При необходимости можно добавить дополнительные методы...
}
```

Реализация нескольких интерфейсов

Класс может реализовать несколько интерфейсов при помощи следующего синтаксиса.

```
public class CDAndDVDComboPlayer : ICDPlayer, IDVDPlayer
```

Если класс реализует несколько интерфейсов, то возможную неоднозначность в именах членов можно разрешить при помощи полного квалификатора имени свойства или метода. Другими словами, производный класс может разрешить конфликт при помощи полного имени метода для указания того, к какому интерфейсу он принадлежит, как в ICDPlayer.Play().

Явная реализация интерфейса

Если класс реализует два интерфейса, содержащих член с одинаковой сигнатурой, то при реализации этого члена в классе оба интерфейса будут использовать этот член для своей реализации. Например:

```
interface IControl
{
    void Paint();
}
interface ISurface
{
    void Paint();
}
class SampleClass : IControl, ISurface
{
    // Как ISurface.Paint так и IControl.Paint вызывают этот метод.
    public void Paint()
    {
    }
}
```

Однако, если члены двух интерфейсов не выполняют одинаковую функцию, это может привести к неверной реализации одного или обоих интерфейсов. Возможна явная реализация члена интерфейса — путем создания члена класса, который вызывается только через интерфейс и имеет отношение только к этому интерфейсу. Это достигается путем включения в имя члена класса имени интерфейса с точкой. Например:

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

Член класса IControl.Paint доступен только через интерфейс IControl, а член ISurface.Paint — только через интерфейс ISurface. Каждая реализация метода является независимой и недоступна в классе напрямую. Например:

```
SampleClass obj = new SampleClass();
//obj.Paint();  // Ошибка компиляции.
IControl c = (IControl)obj;
c.Paint();  // Вызов IControl.Paint класса SampleClass.
ISurface s = (ISurface)obj;
s.Paint(); // Вызов ISurface.Paint класса SampleClass.
```

Явная реализация также используется для разрешения случаев, когда каждый из двух интерфейсов объявляет разные члены с одинаковым именем, например свойство и метод.

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}
```

Для реализации обоих интерфейсов классу необходимо использовать явную реализацию либо для свойства P, либо для метода P, либо для обоих членов, чтобы избежать ошибки компилятора. Например:

```
class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

Явная реализация членов интерфейса

В этом примере выполняется объявление интерфейса IDimensions, и класса Box, который явно реализует члены интерфейса getLength и getWidth. Доступ к этим членам осуществляется через экземпляр интерфейса dimensions.

```
interface IDimensions
{
    float getLength();
    float getWidth();
}
class Box : IDimensions
{
    float lengthInches;
    float widthInches;
    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Явная реализация члена интерфейса:
    float IDimensions.getLength()
    {
        return lengthInches;
    }
    // Явная реализация члена интерфейса:
    float IDimensions.getWidth()
    {
        return widthInches;
    }
    static void Main()
    {
        // Объявление экземпляра класса box1:
        Box box1 = new Box(30.0f, 20.0f);
        // Объявление экземпляра интерфейса dimensions:
        IDimensions dimensions = (IDimensions)box1;
        // Закомментированные строки будут вызывать ошибку компиляции
        // потому что они пытаются получить доступ к явно
реализованному
        // члену интерфейса из экземпляра класса:
        //System.Console.WriteLine("Length: {0}", box1.getLength());
        //System.Console.WriteLine("Width: {0}", box1.getWidth());
        // Вывод размера яшика путем вызова методов
        // из экземпляра интерфейса:
        System.Console.WriteLine("Length: {0}",
dimensions.getLength());
        System.Console.WriteLine("Width: {0}", dimensions.getWidth());
```

```
    }
}
```

- Член интерфейса, реализованный явным образом, недоступен из экземпляра класса.

```
//System.Console.WriteLine("Length: {0}", box1.getLength());
//System.Console.WriteLine("Width: {0}", box1.getWidth());
```

- Следующие строки в методе Main успешно печатают размеры поля, поскольку эти методы вызываются из экземпляра интерфейса.

```
System.Console.WriteLine("Length: {0}", dimensions.getLength());
System.Console.WriteLine("Width: {0}", dimensions.getWidth());
```

Явная реализация членов двух интерфейсов

Явная реализация интерфейса позволяет программисту реализовать два интерфейса, имеющие одинаковые имена членов, и осуществить отдельную реализацию для каждого члена интерфейса. В данном примере отображаются размеры окна как в метрических, так и в британских единицах измерения. Класс "Box" реализует два интерфейса: IEnglishDimensions и IMetricDimensions, которые соответствуют различным системам измерения. Оба интерфейса имеют одинаковые имена членов: "Length" и "Width".

Пример

```
// Объявление интерфейса для английских единиц измерения:
interface IEnglishDimensions
{
    float Length();
    float Width();
}
// Объявление интерфейса для метрических единиц измерения
interface IMetricDimensions
{
    float Length();
    float Width();
}
// Объявление класса Box, который  реализует два интерфейса:
// IEnglishDimensions и IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float lengthInches;
    float widthInches;
    public Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Явная реализация членов IEnglishDimensions:
    float IEnglishDimensions.Length()
    {
        return lengthInches;
    }
    float IEnglishDimensions.Width()
    {
        return widthInches;
    }
    // Явная реализация членов IMetricDimensions:
```

```
    float IMetricDimensions.Length()
    {
        return lengthInches * 2.54f;
    }
    float IMetricDimensions.Width()
    {
        return widthInches * 2.54f;
    }
    static void Main()
    {
        // Объявление экземпляра класса box1:
        Box box1 = new Box(30.0f, 20.0f);
        // Объявление экземпляра интерфейса для английских
        // единиц измерения:
        IEnglishDimensions eDimensions = (IEnglishDimensions)box1;
        // Объявление экземпляра интерфейса для метрических
        // единиц измерения:
        IMetricDimensions mDimensions = (IMetricDimensions)box1;
        // Вывод размеров в английских единицах измерения:
        System.Console.WriteLine("Длина(дюйм): {0}",
eDimensions.Length());
        System.Console.WriteLine("Ширина (дюйм): {0}",
eDimensions.Width());
        // Вывод размеров в метрических единицах измерения:
        System.Console.WriteLine("Длина(см): {0}",
mDimensions.Length());
        System.Console.WriteLine("Ширина (см): {0}",
mDimensions.Width());
    }
}
```

Если необходимо произвести измерения по умолчанию в британских единицах, надо реализовать методы "Length" и "Width" в обычном режиме и явно реализовать методы "Length" и "Width" из интерфейса IMetricDimensions:

```
public float Length()
{
    return lengthInches;
}
public float Width()
{
    return widthInches;
}
// Явная реализация:
float IMetricDimensions.Length()
{
    return lengthInches * 2.54f;
}
float IMetricDimensions.Width()
{
    return widthInches * 2.54f;
}
```

В этом случае можно получить доступ к британским единицам из экземпляра класса, а к метрическим единицам — из экземпляра интерфейса:

```
public static void Test()
```

```
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = (IMetricDimensions)box1;
    System.Console.WriteLine("Длина(дюйм): {0}", box1.Length());
    System.Console.WriteLine("Ширина (дюйм): {0}", box1.Width());
    System.Console.WriteLine("Длина(см): {0}", mDimensions.Length());
    System.Console.WriteLine("Ширина (см): {0}", mDimensions.Width());
}
```