**Практика 12. Наследование**

**1.** Создать интерфейс ICipher, который определяет методы поддержки шифрования строк. В интерфейсе объявляются два метода encode() и decode(), которые используются для шифрования и дешифрования строк, соответственно.

Создать класс ACipher, реализующий интерфейс ICipher. Класс шифрует строку посредством сдвига каждого символа на одну «алфавитную» позицию выше. Например, в результате такого сдвига буква А становится буквой Б.

Создать класс BCipher, реализующий интерфейс ICipher. Класс шифрует строку, выполняя замену каждой буквы, стоящей в алфавите на i-й позиции, на букву того же регистра, расположенную в алфавите на i-й позиции с конца алфавита. Например, буква В заменяется на букву Э.

Написать программу, демонстрирующую функционирование классов.

**2.** Создать класс Figure для работы с геометрическими фигурами. В качестве полей класса задаются цвет фигуры, состояние «видимое/невидимое». Реализовать операции: передвижение геометрической фигуры по горизонтали, по вертикали, изменение цвета, опрос состояния (видимый/невидимый). Метод вывода на экран должен выводить состояние всех полей объекта.

Создать класс Point (точка) как потомок геометрической фигуры. Создать класс Circle (окружность) как потомок точки. В класс Circle добавить метод, который вычисляет площадь окружности. Создать класс Rectangle (прямоугольник) как потомок точки, реализовать метод вычисления площади прямоугольника.

Точка, окружность, прямоугольник должны поддерживать методы передвижения по горизонтали и вертикали, изменения цвета.

Подумать, какие методы можно объявить в интерфейсе, нужно ли объявлять абстрактный класс, какие методы и поля будут в абстрактном классе, какие методы будут виртуальными, какие перегруженными.

**Наследование**

Наследование – это свойство объектно-ориентированной системы наследовать данные и функциональность базового класса. Можно в класс- потомок к методам и полям родительского класса добавить необходимые поля и методы. Класс-потомок может замещать методы родительского класса. Надо помнить, при изменении родительского класса, класс-потомок может оказаться не рабочим.

Наследование от класса называется расширением базового класса. Если класс А наследует от класса В, то класс А называется потомком, а В – предком. Синтаксически это пишется следующим образом:

class A:B {…}

Класс-потомок наследует все элементы базового класса, кроме конструктора и деструктора. Все public элементы базового класса остаются неявно publiс в потомке, private элементы, хоть и наследуются, но доступны только для объектов базового класса. Класс-потомок не может быть более доступным, чем базовый класс.

class Example

{

private class NestedBase { }

public class NestedDerived: NestedBase { } // Ошибка

}

Класс-потомок имеет доступ ко всем protected полям и методам родительского класса, класс не являющийся потомком доступа к protected членам не имеет.

class Token

{

protected string name;

}

class CommentToken:Token

{

public string Name()

{

return name; // Доступ разрешен

}

}

class CommentToken: Token

{

void Fails(Token t)

{

Console.WriteLine(t.name); // Ошибка при компиляции

}

}

Для вызова конструктора базового класса из класса-потомка используется ключевое слово base. Вызов конструктора базового класса пишется после заголовка конструктора класса-потомка через двоеточие:

A(список параметров): base(параметры для передачи в конструктор базового класса)

{

//тело конструктора у класса потомка

}

Ключевое слово base обращается к базовому классу. Для конструктора по умолчанию вызов базового конструктора производится по умолчанию, то есть его можно не прописывать. Для конструкторов не по умолчанию необходимо явно вызывать конструктор базового класса.

class Token

{

protected Token(string name) { ... }

}

class CommentToken: Token

{

public CommentToken(string name) { ... } //ошибка при компиляции

}

В примере выше получим ошибку при компиляции, так как будет вызываться конструктор по умолчанию, но он отсутствует в классе Token. Для правильной работы необходимо записать конструктор в следующем виде:

public CommentToken(string name) : base(name) { ... }

Если конструктор базового класса private, то нельзя создавать конструктор класса потомка.

class NonDerivable

{

private NonDerivable(){ }

}

class Impossible: NonDerivable

{

public Impossible() { } // Ошибка при компиляции

}

В классе-потомке можно переопределять методы базового класса, если они для этого предназначены. Виртуальные методы можно полиморфно переопределять в классах-потомках. В С# по тому, содержит ли базовый класс виртуальные методы, можно определить, был ли этот класс разработан для наследования. Не виртуальный метод имеет только одно определение, одинаковое для всех потомков. Для объявления виртуального метода используется ключевое слово virtual. Виртуальный метод должен содержать тело в базовом классе. Нельзя объявлять виртуальными статичные и private методы. Статичные методы не могут быть виртуальными, так как полиморфизм – это свойство, относящееся к объектам, а не к классам.

Перегруженные методы могут создаваться только для виртуальных методов. Для задания перегруженных методов используется ключевое слово override. В базовом классе метод объявляется виртуальным virtual, в базовом классе есть тело у этого метода, в классе-потомке этот метод определяется перегруженным override и содержит свое тело – таким образом задается перегруженный метод.

class Token

{

public virtual string Name() {…}

}

class CommentToken : Token

{

public override string Name() {…}

}

Можно перегружать только абсолютно идентичные методы. Должны совпадать: имя метода, тип возвращаемого значения, список параметров, уровень доступа. Перегруженный метод должен быть virtual или override. Нельзя объявлять перегруженный метод одновременно виртуальным, т.е. override virtual – нельзя. Нельзя, чтобы перегруженные методы были статичными или private.

Можно скрыть наследуемый метод в иерархии классов, заменив его новым идентичным методом при помощи ключевого слова new. Метод

родительского класса не будет наследоваться потомком, и заменится на новый идентичный метод.

class Token

{

public int LineNumber() {…}

}

class CommentToken : Token

{

new public int LineNumber() {…}

}

Ключевое слово new прячет как виртуальные, так и невиртуальные методы, разрешает проблему совпадения имен, скрывает методы с одинаковой сигнатурой.

В C# можно объявить класс ненаследуемым, при помощи ключевого слова sealed.

public sealed class String {..}

public class MyStr:String; //ошибка – от класса String наследовать нельзя

**Использование интерфейсов**

Интерфейс – синтаксический и семантический шаблон, которого все классы-наследники должны придерживаться. Интерфейс говорит, что он умеет делать, классы определяют, как они это делают. Интерфейс представляет собой класс без какого-либо кода. Все интерфейсы по умолчанию public, модификатор доступа у интерфейсов не используется. У методов также не используется модификатор доступа, по умолчанию методы public. У методов в интерфейсе не должно быть тела, только заголовки методов.

interface IToken

{

public int LineNumber( ){ … }; // Ошибка при компиляции: 1.

Модификатор доступа у метода public //2. Есть тело метода

}

С# позволяет наследовать от одного класса и множества интерфейсов.

Интерфейс может наследовать от многих интерфейсов.

interface IToken { ... } interface IVisitable { ... }

interface IVisitableToken: IVisitable, IToken { ... } class Token: IVisitableToken { ... }

Класс может быть более доступным, чем интерфейс

class Example

{

private interface INested { }

public class Nested: INested { } // Разрешено

}

Класс должен определить все методы всех интерфейсов, от которых он наследует как напрямую, так и косвенно. Метод интерфейса, определяемый классом должен быть идентичен, то есть должны совпадать параметр доступа, имя, возвращаемое значение и список параметров. Интерфейсные методы, реализуемые в классе, могут быть объявлены как virtual. В этом случае классы наследники могут перегружать эти методы в дальнейшем.

Другой способ реализации интерфейсных методов – явная реализация. При явной реализации необходимо указать полное имя метода: Имя\_интерфейса.имя\_метода. При явном определении метод не может быть виртуальным, должен отсутствовать модификатор доступа. При вызове метода к нему нет прямого доступа, только через интерфейс.

class Token: IToken, IVisitable

{

string IToken.Name( )

{

...

}

private void Example( )

{

Name( ); // Ошибка при компиляции ((IToken)this).Name( ); // Правильно

}

...

}

Явная реализация позволяет:

* исключить определение интерфейса из класса, если он не интересен пользователям класса.
* обеспечивать классу несколько определений различных методов интерфейсов одинаковой сигнатуры.

interface IArtist

{

void Draw();

}

interface ICowboy

{

void Draw();

}

class ArtisticCowboy: IArtist, ICowboy

{

void IArtist.Draw() {…} void ICowbowt.Draw() {…}

}

**Использование абстрактных классов**

Абстрактные классы используются для частичной реализации классов, которые могут быть полностью реализованы в конкретных классах-потомках. Абстрактный класс объявляется с помощью ключевого слова abstract. Правила создания абстрактного класса совпадают с правилами создания обычных классов. Однако, в абстрактных классах можно объявлять абстрактные методы. Нельзя создавать объекты абстрактного класса. Абстрактный класс может являться наследником неабстрактного класса. Все методы интерфейса, определяемого абстрактным классом, должны быть определены в абстрактном классе.

И абстрактные классы, и интерфейсы предназначены для наследования. Однако класс может наследовать только от одного абстрактного класса. Только абстрактные классы могут иметь абстрактные методы. У абстрактного метода отсутствует тело метода. Абстрактные методы – виртуальные, переопределенные абстрактные методы у классов-потомков будут override. Абстрактные методы могут переопределять virtual и override методы.

class Token

{

public virtual string Name( ) { ... }

}

abstract class Force: Token

{

public abstract override string Name( );

}