Лабораторная работа №2

«Наследование, модификаторы доступа, ООП»

**Цели работы:**

1. Научиться синтаксису и основным принципам наследования в C#.

**Задание№1**

Создайте класс, представляющий учебный класс ClassRoom.

Создайте класс ученик - Pupil.

В теле класса создайте методы void Study(), void Read(), void Write(), void Relax().

Создайте 3 производных класса ExcelentPupil, GoodPupil, BadPupil от класса базового класса Pupil и переопределите каждый из методов, в зависимости от успеваемости ученика (реализация может быть произвольной, например простой вывод на консоль разных строк).

Конструктор класса ClassRoom принимает аргументы типа Pupil, класс должен состоять из 4 учеников.

Предусмотрите возможность того, что пользователь может передать 2 или 3 аргумента.

Выведите информацию о том, как все ученики экземпляра класса ClassRoom умеют учиться, читать, писать, отдыхать.

Примечание: при реализации возможности создания экземпляра класса ClassRoom с произвольным количеством учеников воспользуйтесь ключевым словом params.

**Задание№2**

Создайте класс vehicle.

В теле класса создайте поля: координаты и параметры средств передвижения (цена, скорость, год выпуска).

Создайте 3 производных класса Plane, Саг и Ship.

Для класса Plane должна быть определена высота и количество пассажиров.

Для класса Ship — количество пассажиров и порт приписки.

Написать программу, которая выводит на экран информацию о каждом средстве передвижения.

Примечание: избегайте дублирования кода, используйте ключевое слово base после объявления конструкторов в классах наследниках для вызова и передачи параметров в конструктор базового класса.

**Задание №3**

Создайте класс DocumentWorker.

В теле класса создайте три метода OpenDocument(), EditDocument(), SaveDocument().

В тело каждого из методов добавьте вывод на экран соответствующих строк: "Документ открыт", "Редактирование документа доступно в версии Pro", "Сохранение документа доступно в версии Pro".

Создайте производный класс ProDocumentWorker.

Переопределите соответствующие методы, при переопределении методов выводите следующие строки: "Документ отредактирован", "Документ сохранен в старом формате, сохранение в остальных форматах доступно в версии Expert".

Создайте производный класс ExpertDocumentWorker от базового класса ProDocumentWorker.

Переопределите соответствующий метод. При вызове данного метода необходимо выводить на экран "Документ сохранен в новом формате".

В теле метода Main() реализуйте возможность приема от пользователя номера ключа доступа pro и exp.

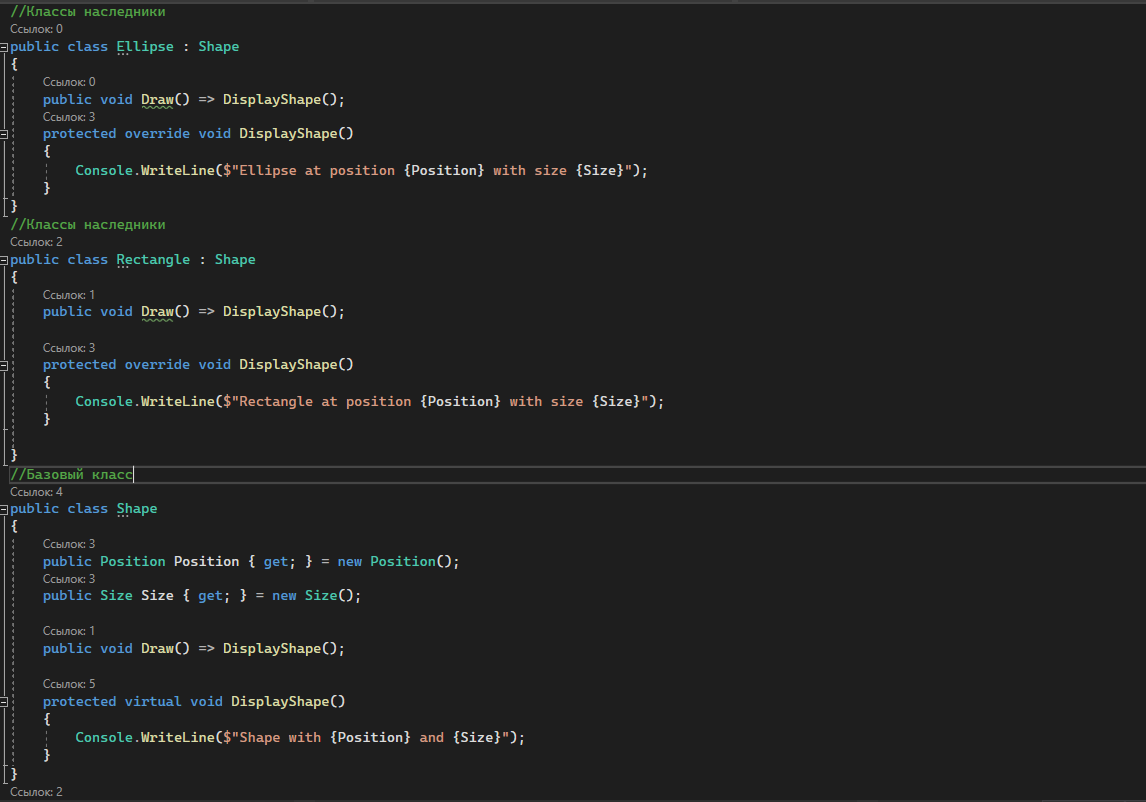
Если пользователь не вводит ключ, он может пользоваться только бесплатной версией (создается экземпляр базового класса), если пользователь ввел номера ключа доступа pro и exp, то должен создаться экземпляр соответствующей версии класса, приведенный к базовому – DocumentWorker.

**Теоретические сведения**

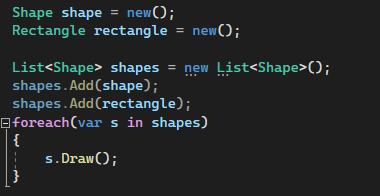
**Принципы ООП**

* **Полиморфизм** означает способность языка трактовать связанные объекты в сходной манере. В частности, этот принцип ООП позволяет базовому классу определять набор членов (формально называемый полиморфным интерфейсом), которые доступны всем наследникам. Полиморфный интерфейс класса конструируется с использованием любого количества виртуальных или абстрактных членов. В более общем смысле понятие полиморфизма нередко выражается следующим образом: "один интерфейс — множество методов". Это означает, что для группы взаимосвязанных действий можно разработать общий интерфейс. Полиморфизм помогает упростить программу, позволяя использовать один и тот же интерфейс для описания общего класса действий.
* **Наследование** касается способности языка позволять строить новые определения классов на основе определений существующих классов. По сути, наследование позволяет расширять поведение базового (или родительского) класса, наследуя основную функциональность в производном подклассе (также именуемом дочерним классом) Т.е. наследование представляет собой процесс, в ходе которого один объект приобретает свойства другого объекта. Это очень важный процесс, поскольку он обеспечивает принцип иерархической классификации.
* **Инкапсуляция** это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. В объектно-ориентированном языке данные и код могут быть объединены в совершенно автономный черный ящик. Внутри такого ящика находятся все необходимые данные и код. Когда код и данные связываются вместе подобным образом, создается объект. Иными словами, объект — это элемент, поддерживающий инкапсуляцию. Т.е. инкапсуляция представляет собой способности языка скрывать излишние детали реализации от пользователя объекта. Основной единицей инкапсуляции в C# является класс, который определяет форму объекта. Он описывает данные, а также код, который будет ими оперировать. В C# описание класса служит для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта.
* **Абстрагирование** это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Абстракция — это использование только тех характеристик объекта, которые с достаточной точностью представляют его в данной системе. Основная идея состоит в том, чтобы представить объект минимальным набором полей и методов и при этом с достаточной точностью для решаемой задачи. Абстракция присуща не только ООП, поэтому ее не всегда включают в список основных концепций.
* **Агрегация** это то, что можно объединить с объектом. Например, человек не является частью объекта «Автомобиль», но он может сесть на сиденье водителя и затем стать водителем автомобиля — два отдельных объекта, которые объединяются вместе, чтобы сформировать новый компонент.
* **Композиция** – это то, из чего сделан предмет. Например, автомобиль состоит из разных частей, таких как четыре объекта «Колесо», несколько объектов «Сиденье» и двигатель.

В языке C# наследование синтаксически реализуется указанием символа “двоеточие” после объявления имени класса:



Теперь, к примеру, можно объявлять коллекции базового типа и наполнять их элементами производного типа, что позволит вызывать на объектах методы с одинаковым названием, но разной реализацией:



Результат выполнения:

