Абстрактные классы, интерфейсы, делегаты

Иногда требуется описать базовый класс, экземпляры которого создавать не нужно. В этом классе определяется самая общая форма для всех его производных классов. Предполагается, что наполнять эту форму деталями будут производные классы.

Например, объявим класс фигуры, который будет хранить такие значения, как левая и правая позиции фигуры на форме, и её имя. Также можно объявить метод, который будет рисовать фигуру:

```
abstract class Figure
{
    public int left { get; set; }
    public int top { get; set; }

    abstract public void Drow();
}
```

Ключевое слово `abstract` говорит о том, что нельзя создавать непосредственно объекты этого класса. Описанный класс нужен лишь для того, чтобы объявить в нём свойства и методы, которые понадобятся наследникам.

```
class RectangleFigure : Figure
{
    public int Width { get; set; }
    public int Height { get; set; }

    public override void Drow()
  {
    Console.WriteLine("Это класс прямоугольника");
  }
}

class CircleFigure : Figure
{
    public int Radius { get; set; }

    public override void Drow()
  {
    Console.WriteLine("Это класс круга");
  }
}
```

Оба класса являются наследниками `Figure`, наследуют от предка свойства `left` и `top`, а также реализуют его метод `Drow()`. Теперь можно создавать их объекты, и даже больше: можно создавать их как переменные класса `Figure`.

```
class Program
{
    static void Main(string[] args)
    {
        Figure rect;
        rect = new RectangleFigure();
        rect.Drow();

        rect = new CircleFigure();
        rect.Drow();
    }
}
```

Приведём пример использования ключевых слов `is` и `as`.

В этом примере с помощью `is` проверяем, является ли объект `shape` представителем класса `CircleFigure`:

```
Figure shape = new CircleFigure();
if (shape is CircleFigure) Console.WriteLine(Radius);
```

Слово `as` позволяет приводить типы классов:

```
Figure shape = new CircleFigure();
CircleFigure circle = (CircleFigure) shape;
```

Тоже самое, с помощью `as`:

```
Figure shape = new CircleFigure();
CircleFigure circle = shape as CircleFigure;
```

Интерфейсы.

Интерфейс — это объявление, схожее с (абстрактным) классом, в нём нет реализаций методов, т.е. методы его абстрактны. С их помощью можно описать, какими функциями должен обладать класс и что он должен уметь делать. При этом интерфейс не имеет самого кода и не реализует функции, кроме того, нельзя указывать модификаторы доступа для описываемых методов — они все считаются открытыми.
Вернёмся к классу `Person` из работы №2.

```
namespace ClassPerson
{
    class Person
    {
        public Person(string firstName, string lastName)
        {
            FirstName = firstName;
            LastName = lastName;
        }

        public string FirstName { get; set; }
        public string LastName { get; set; }
   }
}
```

Наделим человека функциями хранения денег. Деньги могут храниться в кошельке или в сейфе. Это разные по идеологии классы объектов, поэтому пытаться реализовать их из одного класса неправильно, намного эффективнее описать функции, которые нужны для работы с деньгами в интерфейсе, и наследовать интерфейс в классе.
Опишем интерфейс кошелька, лучше это делать в отдельном файле, но можно добавить описание в файл с классом, просто расположив их рядом в одном пространстве имён:

```
namespace ClassPerson
{
    interface Ipurse
    {
        int Sum
        {
            get;
            set;
        }
        void AddMoney(int sum);
```

```
        int DecMoney(int sum);
    }
}
```

Поскольку интерфейс не имеет реализации методов, а только объявляет их, то нельзя создать экземпляр интерфейса. Интерфейсы нужны для их реализации в классах.
Наделим нашего человека кошельком, для этого наследуем интерфейс:

```
class Person: IPurse
    {
      //реализация класса
    }
```

Если вы наследуете класс и интерфейсы, то имя класса в списке предков указывают первым. Опишем внутри своего класса реализацию всех свойств и методов интерфейса (они обязательно должны быть открытыми):

```
class Person: IPurse
    {
      //реализация класса
      ...

      // реализация интерфейса
        int sum = 0;
        public int Sum
        {
            get { return sum; }
            set { sum = value; }
        }

        public void AddMoney(int sum)
        {
            Sum += sum;
        }

        public int DecMoney(int sum)
        {
            Sum -= sum;
            return Sum;
        }
    }
```

Теперь посмотрим, как можно использовать интерфейсы. Создайте windows -приложение, расположите на форме две кнопки, одна — для добавления денег в кошелёк, другая — для снятия. Для отображения остатка поместите на форму метку `Label`. Сумма, которая будет сниматься или добавляться в кошелёк, будет вводиться через поле ввода `NumericUpDown`.
Код приложения представлен ниже.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace PersonInterface
{
```

```
    public partial class Form1 : Form
    {
        //объявляем переменные класса
        Person person = new Person("Абрам", "Абрамович");
        Object personObject;
        Ipurse purse;

        public Form1()
        {
            InitializeComponent();
            //инициализация переменных
            personObject = person;
            purse = person;
        }

        //добавление денег в кошелёк
        private void buttonAdd_Click(object sender, EventArgs e)
        {
            if (personObject is Ipurse)
            {
                ((Ipurse)personObject).AddMoney((int)numericUpDown1.Value);
                label1.Text = ((Ipurse)personObject).Sum.ToString();
            }
        }

        private void buttonDec_Click(object sender, EventArgs e)
        {
            purse.DecMoney((int)numericUpDown1.Value);
            label1.Text = purse.Sum.ToString();
        }
    }
}
```

В начале класса мы описали три переменные типов `Person, Object, Ipurse.` В нашем приложении все три переменные, несмотря на разный класс, имеют одно и то же значение, и мы сможем работать через них с одним и тем же объектом. В методе добавления денег приведён пример работы с переменной класса `Object.` А в методе уменьшения денег мы напрямую вызываем методы интерфейса через интерфейсную переменную `purse.`

Таким образом, **абсолютно всё равно, объект какого класса находится в переменной — главное, чтобы этот объект реализовывал интерфейс, и тогда его метод будет вызван корректно.**

Задание. Проанализируйте код приложения, и перепишите код, реализовав работу с переменными типа `Person, Object, Ipurse.` В результате должно получиться три приложения, в каждом из которых используется только одна переменная. Покажите преподавателю.

Интерфейсы в качестве параметров.

Интерфейсы удобны не только с точки зрения унификации доступа к данным, но и для получения универсального типа данных, который можно передавать как переменные в другие методы.

В нашем примере можно использовать метод, который будет универсально уменьшать деньги в кошельке:

```
 void DecMoney(Ipurse purse)
        {
            purse.DecMoney((int)numericUpDown1.Value);
```

```
            label1.Text = purse.Sum.ToString();
        }

 private void buttonDec_Click(object sender, EventArgs e)
        {
            DecMoney(person);
        }
```

Наследование интерфейсов.

Интерфейсы тоже позволяют наследование, и оно работает точно так же, как у классов. Если интерфейс наследует какой-либо другой интерфейс, то он наследует все его метод и свойства. Например, введём интерфейс сейфа:

```
interface ISafe : Ipurse
    {
        bool Locked
        {
            get;
        }
        void Lock;
        void Unlock;
    }
```

Этот интерфейс наследует методы кошелька, только в данном случае они будут класть деньги не в кошелёк, а в сейф и забирать их оттуда. Помимо этого, интерфейс имеет методы открытия `Unlock` и закрытия `Lock` сейфа, а также булево значение `Locked.` которое будет возвращать значение `true`, когда сейф закрыт.

Задание 2. Реализуйте интерфейс `ISafe` в классе `Person.` Создайте приложение, демонстрирующее работу с методами интерфейса.

Задание 3. Дополните класс `Person` возможностью хранения списка детей для каждой персоны. Список детей возможно хранить с помощью динамического массива (работа с ними описана в пространстве имён `System.Collections`), например, так:

```
        ArrayList Children = new ArrayList();

        public void AddChild(string firstName, string lastName)
        {
            Children.Add(new Person(firstName,lastName));
        }

        public void DeleteChild(int index)
        {
            Children.RemoveAt(index);
        }

        public Person GetChild(int index)
        {
            return (Person)Children[index];
        }
```

Создайте приложение, демонстрирующее работу со списками детей: возможность добавлять, удалять ребёнка из списка, получать количество детей каждой персоны, получать поимённый список детей персоны.

Стандартные интерфейсы .NET

Интрефейс IComparable

Большинство встроенных в .NET классов коллекций и массивы поддерживают сортировку. С помощью одного метода, который, как правило, называется Sort() можно сразу отсортировать по возрастанию весь набор данных.

Однако метод Sort по умолчанию работает только для наборов примитивных типов, как int или string. Для сортировки наборов сложных объектов применяется интерфейс **IComparable**. Он имеет всего один метод, возвращающий результат сравнения двух объектов — текущего и переданного ему в качестве параметра:

```
interface  IComparable
{
      int  CompareTo(  object  obj  )
}
```

Метод должен возвращать:

- 0, если текущий объект и параметр равны;
- отрицательное число, если текущий объект меньше параметра;
- положительное число, если текущий объект больше параметра.

Реализуем интерфейс **IComparable** в классе **Monster**. В качестве критерия сравнения объектов выберем поле health. В листинге приведена программа, сортирующая массив монстров по возрастанию величины, характеризующей их здоровье.

```
using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace Monster_IComparable
{
    class Monster: IComparable
    {
        public Monster(int health, int ammo, string name)
        {
            this.health = health;
            this.ammo = ammo;
            this.name = name;
        }

// Перегрузка методов, унаследованных от Object,
//с целью использовать их при перегрузке операций:

        public override bool Equals(object obj)
        {
            if (obj == null || GetType()!=obj.GetType()) return false;

            Monster temp = (Monster)obj;
                return health == temp.health && ammo == temp.ammo && name ==
temp.name;
        }

        public override int GetHashCode()
        {
            return name.GetHashCode();
        }
```

```
// Реализуем метод интерфейса:

        public int CompareTo(object obj)
        {
            Monster temp = (Monster)obj;
            if (this.health > temp.health) return 1;
            if (this.health < temp.health) return -1;
            return 0;
        }

        public static bool operator ==(Monster a, Monster b)
        {
            return a.Equals(b);
        }

        public static bool operator !=(Monster a, Monster b)
        {
            return ! a.Equals(b);
        }

//Можем использовать реализацию CompareTo() для перегрузки операций:

        public static bool operator < (Monster a, Monster b)
        {
            return (a.CompareTo(b)<0);
        }

        public static bool operator >(Monster a, Monster b)
        {
            return (a.CompareTo(b) > 0);
        }

        public static bool operator <=(Monster a, Monster b)
        {
            return (a.CompareTo(b) <= 0);
        }

        public static bool operator >=(Monster a, Monster b)
        {
            return (a.CompareTo(b) >= 0);
        }

        public void Passport()
        {
            Console.WriteLine("Monster {0} \t health = {1} \t ammo = {2}", name,
health, ammo);
        }

        string name;
        int health, ammo;
    }

    class Program
    {
        static void Main()
        {
            Monster Вася = new Monster(70, 80, "Вася");
            Monster Петя = new Monster(80, 80, "Петя");

            Console.WriteLine( "Сравнение монстров:" );
            if       (Вася > Петя) Console.WriteLine("Вася больше Пети");
```

```
            else
                  if (Вася == Петя) Console.WriteLine("Петя как Вася");
                  else Console.WriteLine("Вася меньше Пети");
            Console.WriteLine();

            Monster[] ArrMonstr = new Monster[3];
            ArrMonstr[0] = new Monster(70, 80, "Вася");
            ArrMonstr[1] = new Monster(80, 80, "Петя");
            ArrMonstr[2] = new Monster(90, 90, "Маша");

      // Теперь можем отсортировать массив, благодаря реализации интерфейса:
            Array.Sort(ArrMonstr);
            foreach (Monster mons in ArrMonstr) mons.Passport();
            Console.ReadKey();
        }
    }
}
```

Интрефейс IComparer
Сортировка по разным критериям

Интерфейс IComparer также определен в пространстве имен System.Collections. Он содержит один метод Compare, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

```
interface  IComparer
{
      int  Compare  (  object  obi,  object  ob2  )
}
```

Принцип применения этого интерфейса состоит в том, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий этот интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента (существует несколько перегруженных версий этого метода).

Пример сортировки массива объектов из предыдущего листинга по именам (свойство Name, класс SortByName) и количеству вооружений (свойство Ammo, класс SortByAmmo) приведен в следующем листинге. Классы параметров сортировки объявлены вложенными, поскольку они требуются только объектам класса Monster.

```
using System;
using System.Collections;
namespace Monster_IComparer
{
    class Monster
    {
        public Monster(int health, int ammo, string name)
        {
            this.health = health;
            this.ammo = ammo;
            this.name = name;
        }

        public int Ammo
        {
            get { return ammo; }
            set
            {
                if (value > 0) ammo = value;
```

```
                else ammo = 0;
            }
        }
        public string Name
        {
            get { return name; }
        }
        virtual public void Passport()
        {
            Console.WriteLine("Monster {0} \t health = {1} ammo = {2}",
                              name, health, ammo);
        }

// класс для сортировки по имени
        public class SortByName : IComparer
        {
            int IComparer.Compare(object ob1, object ob2)
            {
                Monster m1 = (Monster)ob1;
                Monster m2 = (Monster)ob2;
                return String.Compare(m1.Name, m2.Name);
            }
        }

// класс для сортировки по оружию
        public class SortByAmmo : IComparer
        {
            int IComparer.Compare(object ob1, object ob2)
            {
                Monster m1 = (Monster)ob1;
                Monster m2 = (Monster)ob2;
                if (m1.Ammo > m2.Ammo) return 1;
                if (m1.Ammo < m2.Ammo) return -1;
                return 0;
            }

        }

        string name;
        int ammo, health;
    }


class Program
        {
            static void Main()
            {
                const int n = 3;
                Monster[] stado = new Monster[n];
                stado[0] = new Monster(50, 50, "Вася");
                stado[1] = new Monster(80, 80, "Петя");
                stado[2] = new Monster(40, 10, "Маша");

                Console.WriteLine("Сортировка по имени:");

// передаём в метод Sort() вторым параметром объект класса SortByName:

                Array.Sort(stado, new Monster.SortByName());
                foreach (Monster elem in stado) elem.Passport();

                Console.WriteLine("Сортировка по вооружению:");
```

```
// передаём в метод Sort() вторым параметром объект класса SortByAmmo:

                Array.Sort(stado, new Monster.SortByAmmo());
                foreach (Monster elem in stado) elem.Passport();
                Console.ReadKey();
            }
        }
    }
```

Интрефейс IClonable
Клонирование

При простом присваивании переменных ссылочных типов, как в листинге:

```
Object p1 = new Object();
p2 = p1;
```

имена переменных становятся псевдонимами, т.е. p1 и p2 будут указывать на один и тот же объект в памяти, поэтому изменения свойств в переменной p2 затронут также и переменную p1.

Чтобы переменная p2 указывала на новый объект, но со значениями из p1, мы можем применить клонирование с помощью реализации интерфейса ICloneable:

```
public interface ICloneable
{
    object Clone();
}
```

Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом MemberwiseClone(), который любой объект наследует от класса object. При этом объекты, на которые указывают поля объекта, в свою очередь являющиеся ссылками, не копируются. Это называется поверхностным клонированием.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MonstersIClonable
{
    class Company
    {
        public string NameCompany { get; set; }
    }
    class Monster : ICloneable
    {
        public Monster ShallowClone()           // поверхностная копия
        { return (Monster)this.MemberwiseClone(); }

        public object Clone()               // пользовательская копия
        {
            return new Monster
            {
                health = this.health,
                ammo = this.ammo,
                name = "Клон " + this.name,
                company = this.company
```

```
            };
        }
    }


        virtual public void Passport()
        {
            Console.WriteLine("Monster {0}\thealth = {1} ammo = {2} company =
                                {3}",name, health, ammo, company.NameCompany);
        }
        public string name { get; set; }
        public int health { get; set; }
        public int ammo { get; set; }
        public Company company;
    }

class Program
    {
        static void Main()
        {
            Monster Вася = new Monster {
                health = 100,
                ammo = 100,
                name = "Вася",
                company = new Company { NameCompany = "Crazy Inc" } };

            // псевдоним:
            Monster X = Вася;
            // поверхностный клон:
            Monster Y = Вася.ShallowClone();
            // пользовательский клон:
            Monster Z = (Monster)Вася.Clone();

            Вася.Passport();
            X.Passport();
            Y.Passport();
            Z.Passport();
            Console.ReadLine();

            Вася.health = 10;
            Вася.company.NameCompany = "Cool Inc";
            Вася.Passport();
            X.Passport();
            Y.Passport();
            Z.Passport();
            Console.ReadLine();

        }
    }
}
```

Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять **глубокое копирование**:

```
class Monster : ICloneable
    {
      ...
        public object FullClone()               //глубокое клонирование
        {
```

```
            Company company = new Company { NameCompany = this.company.NameCom-
pany };
            return new Monster
            {
                health = this.health,
                ammo = this.ammo,
                name = "Клон " + this.name,
                company = company
            };
        }
    }

class Program
    {
        static void Main()
        {
            …

            Monster FullClon = (Monster)Вася.FullClone();
            Вася.company.NameCompany = "MegaCool Inc";
            Вася.Passport();
            FullClon.Passport();
            Console.ReadLine();
      }
}
```

Делегаты

Делегат (delegate) — это тип, который позволяет хранить ссылки на функции. Объявляются делегаты практически также, как и функции, но только безо всякого тела функции и с ключевым словом **delegate.**

В объявлении любого делегата указывается возвращаемый тип и список параметров.

`[спецификаторы ] delegate тип имя_делегата ( [ параметры ] )`

Спецификаторы делегата имеют тот же смысл, что и для класса, причем допускаются только спецификаторы `new, public, protected, internal` и `private.`

`Тип` – тип функции, `параметры` – параметры функции.

```
public delegate void D ( int i );
```

После определения делегата можно объявлять переменную с типом этого делегата. Далее эту переменную можно инициализировать как ссылку на любую функцию, которая имеет точно такой же возвращаемый тип и список параметров, как и у делегата. После этого функцию можно вызывать с использованием переменной делегата так, будто бы это и есть сама функция.

Многоадресатная передача — это способность создавать список вызовов (или цепочку вызовов) методов, которые должны автоматически вызываться при вызове делегата. Для этого достаточно создать экземпляр делегата, а затем для добавления методов в эту цепочку использовать оператор "+=". Для удаления метода из цепочки используется оператор " - = " (можно + и – соответственно). Делегат с многоадресатной передачей имеет одно ограничение: он должен возвращать тип void.

События

События похожи на исключения тем, что они тоже генерируются, т.е. выдаются объектами, и тем, что для них тоже можно предоставлять реагирующий на них выполнением какого-

нибудь действия код. Однако существует и несколько отличий, наиболее важное из которых состоит в отсутствии для обработки событий структуры, эквивалентной try. . . catch. Вместо применения этой структуры на события нужно подписываться (subscribe).
Под подпиской на событие подразумевается предоставление кода, который должен выполняться при генерации данного события, в виде обработчика событий (event handler).
На событие можно подписывать несколько обработчиков, которые тогда все будут вызываться при генерации этого события. Эти обработчики могут являться как частью того класса объекта, который генерирует данное событие, так и частью других классов.
Сами обработчики событий представляют собой просто функции. Единственным ограничением для такой функции является то, что ее возвращаемый тип и параметры должны обязательно соответствовать тем, которых требует событие. Это ограничение входит в состав определения события и задается **делегатом**.
Базовая последовательность обработки выглядит следующим образом:

1 Приложение создает объект, который может генерировать событие.
2 Приложение подписывается на событие.
3 При генерации события подписчику отправляется соответствующее уведомление.

Перед определением события требуется обязательно определить используемый вместе с событием тип делегата, т.е. тип делегата, типу и параметрам которого должен соответствовать метод обработки событий. Для выполнения этого используется стандартный синтаксис делегатов, с помощью которого необходимый делегат определяется как общедоступный.

Для обработки события на него нужно подписываться, предоставляя функцию — обработчик событий, возвращаемый тип и параметры которой должны совпадать с возвращаемым типом и параметрами делегата, закрепленного для применения с этим событием.

Пример программы, обрабатывающей события

В C# каждое событие определяется делегатом, описывающим сигнатуру сообщения.
Объявление события - это двухэтапный процесс:

- Объявляется делегат - функциональный класс, задающий сигнатуру.

- В классе, создающем события, объявляется событие как экземпляр соответствующего делегата**.**

```
delegate void CollectionHandler(object source,
CollectionHandlerEventArgs args);//делегат
    class MyNewCollection:MyCollection
    {
//происходит при добавлении нового элемента или при удалении элемента из
//коллекции
 public event CollectionHandler CollectionCountChanged;
//объекту коллекции присваивается новое значение
public event CollectionHandler CollectionReferenceChanged;
```

- Поскольку действия по включению могут повторяться, полезно в состав методов класса добавить защищенную процедуру, включающую событие. Даже если событие генерируется только в одной точке, написание такой процедуры считается признаком хорошего стиля. Этой процедуре обычно дается имя, начинающееся со слова On, после которого следует имя события. Будем называть такую процедуру On-процедурой. Она проста и состоит из вызова объявленного события, включенного в тест, который

проверяет перед вызовом, а есть ли хоть один обработчик события, способный принять соответствующее сообщение.

```
//обработчик события CollectionCountChanged
public virtual void OnCollectionCountChanged(object source,
CollectionHandlerEventArgs args)
        {
            if (CollectionCountChanged != null)
               CollectionCountChanged(source, args);
        }
//обработчик события OnCollectionReferenceChanged
public virtual void OnCollectionReferenceChanged(object source,
CollectionHandlerEventArgs args)
       {
            if (CollectionReferenceChanged != null)
                CollectionReferenceChanged(source, args);
        }
```

Объекты, которые принимают сообщение о событии, должны заранее присоединить обработчики событий к объекту EventHandler evnt, задающему событие.

- Последний шаг, который необходимо выполнить в классе создающем события - это в нужных методах класса вызвать процедуру On. Естественно, что перед вызовом нужно определить значения входных аргументов события. После вызова может быть выполнен анализ выходных аргументов, определенных обработчиками события.

```
      public override bool Remove(int position)
{
 OnCollectionCountChanged(this, new
CollectionHandlerEventArgs(this.Name, "delete", list[position]));
 return base.Remove(position);
 }
     public override int Add(Person p)
 {
 OnCollectionCountChanged(this, new
CollectionHandlerEventArgs(this.Name, "add", p));
return base.Add(p);
 }
     public override Person this[int index]
        {
          get
            {
               return base[index];
            }
          set
            {
OnCollectionReferenceChanged(this, new
CollectionHandlerEventArgs(this.Name, "changed", list[index]));
base[index] = value;
            }
         }
```

Объекты класса-отправителя создают события и уведомляют о них объекты класса (классов)-получателя событий.

Класс-получатель должен иметь обработчик события – процедуру, согласованную по сигнатуре с функциональным типом делегата, который задает событие;

```
  public void CollectionCountChanged(object sourse,
CollectionHandlerEventArgs e)
        {
            JournalEntry je = new JournalEntry(e.NameCollection,
e.ChangeCollection, e.Obj.ToString());
            journal.Add(je);

        }
       public void CollectionReferenceChanged(object sourse,
CollectionHandlerEventArgs e)
        {
            JournalEntry je = new JournalEntry(e.NameCollection,
e.ChangeCollection, e.Obj.ToString());
            journal.Add(je);
        }
```

- Подписка на события заключается в присоединении обработчика события к event-объекту:

```
MyNewCollection mc1 = new MyNewCollection("FIRST");
//один объект Journal подписать на события CollectionCountChanged и
CollectionReferenceChanged из первой коллекции
            Journal joun1 = new Journal();
            mc1.CollectionCountChanged += new
CollectionHandler(joun1.CollectionCountChanged);
            mc1.CollectionReferenceChanged += new
CollectionHandler(joun1.CollectionReferenceChanged);
```

Задания для самостоятельной работы

Задание 1

Замечания. 1) Полную структуру классов и их взаимосвязь продумать самостоятельно. 2) Для абстрактного класса определить, какие методы должны быть абстрактными, а какие обычными.

Вариант 1

1. Создать абстрактный класс Figure с функциями вычисления площади и периметра, а также функцией, выводящей информацию о фигуре на экран.
2. В абстрактном классе Figure реализовать метод CompareTo так, чтобы можно было отсортировать объекты по их площадям.
3. Создать производные классы: Rectangle (прямоугольник), Circle (круг), Triangle (треугольник).
4. Создать обобщенный список n фигур и вывести полную информацию о фигурах на экран, отсортировав объекты по их площадям.

Вариант 2

1. Создать абстрактный класс Function с функциями вычисления значения по формуле y=f(x) в заданной точке, а также функцией, выводящей информацию о виде функции на экран.
2. В абстрактном классе Function реализовать метод CompareTo так, чтобы можно было отсортировать функции по коэффициенту a.
3. Создать производные классы: Line (y=ax+b), Kub (y=ax2 +bx+c), Hyperbola (y=a/x).
4. Создать обобщенный список n функций и вывести полную информацию о значении данных функций в точке x, отсортировав функции по коэффициенту a.

Вариант 3

1. Создать абстрактный класс Edition с функциями, позволяющими вывести на экран информацию об издании, а также определить, является ли данное издание искомым.
2. В абстрактном классе Edition реализовать метод CompareTo так, чтобы можно было отсортировать каталог изданий по фамилии автора.
3. Создать производные классы: Book (название, фамилия автора, год издания, издательство), Article (название, фамилия автора, название журнала, его номер и год издания), OnlineResource (название, фамилия автора, ссылка, аннотация).
4. Создать каталог (обобщенный список) из n изданий, вывести полную информацию из каталога, отсортировав каталог изданий по фамилии автора, а также организовать поиск изданий по фамилии автора.

Вариант 4

1. Создать абстрактный класс Transport с функциями, позволяющими вывести на экран информацию о транспортном средстве, а также определить грузоподъемность транспортного средства.
2. В абстрактном классе Transport реализовать метод CompareTo так, чтобы можно было отсортировать базу данных о машинах по их грузоподъемности.
3. Создать производные классы: Car (марка, номер, скорость, грузоподъемность), Motorbike (марка, номер, скорость, грузоподъемность, наличие коляски, при этом если коляска отсутствует, то грузоподъемность равна 0), Truck (марка, номер, скорость, грузоподъемность, наличие прицепа, при этом если есть прицеп, то грузоподъемность увеличивается в два раза).
4. Создать базу (обобщенный список) из n машин, вывести полную информацию из базы на

экран, отсортировав базу данных о машинах по их грузоподъемности, а также организовать поиск машин, удовлетворяющих требованиям грузоподъемности.

Вариант 5

1. Создать абстрактный класс Persona с функциями, позволяющими вывести на экран информацию о персоне, а также определить ее возраст (на момент текущей даты).
2. В абстрактном классе Persona реализовать метод CompareTo так, чтобы можно было отсортировать базу данных о персонах по дате рождения.
3. Создать производные классы: Enrollee (фамилия, дата рождения, факультет), Student (фамилия, дата рождения, факультет, курс), Teacher (фамилия, дата рождения, факультет, должность, стаж).
4. Создать базу (обобщенный список) из n персон, вывести полную информацию из базы на экран, отсортировав базу данных о персонах по дате рождения, а также организовать поиск персон, чей возраст попадает в заданный диапазон.

Вариант 6

1. Создать абстрактный класс Goods с функциями, позволяющими вывести на экран информацию о товаре, а также определить, соответствует ли он сроку годности на текущую дату.
2. В абстрактном классе Goods реализовать метод CompareTo так, чтобы можно было отсортировать базу данных о товарах по их цене.
3. Создать производные классы: Product (название, цена, дата производства, срок годности), Party (название, цена, количество шт, дата производства, срок годности), Kit (название, цена, перечень продуктов).
4. Создать базу (обобщенный список) из n товаров, вывести полную информацию из базы на экран, отсортировав базу данных о товарах по их цене, а также организовать поиск просроченного товара (на момент текущей даты).

Вариант 7

1. Создать абстрактный класс Goods с функциями, позволяющими вывести на экран информацию о товаре, а также определить, соответствует ли она искомому типу.
2. В абстрактном классе Goods реализовать метод CompareTo так, чтобы можно было отсортировать базу данных о товарах по возрасту детей, на которых он рассчитан.
3. Создать производные классы: Toy (название, цена, производитель, материал, возраст, на который рассчитана), Book (название, автор, цена, издательство, возраст, на который рассчитана), SportsEquipment (название, цена, производитель, возраст, на который рассчитан).
4. Создать базу (обобщенный список) из n товаров, вывести полную информацию из базы на экран, отсортировав базу данных о товарах по возрасту детей, на которых он рассчитан, а также организовать поиск товаров определенного типа.

Вариант 8

1. Создать абстрактный класс TelephoneDirectory с функциями, позволяющими вывести на экран информацию о записях в телефонном справочнике, а также определить соответствие записи критерию поиска.
2. В абстрактном классе TelephoneDirectory реализовать метод CompareTo так, чтобы можно было отсортировать базу данных справочника по номеру телефона.
3. Создать производные классы: Persona (фамилия, адрес, номер телефона), Organization (название, адрес, телефон, факс, контактное лицо), Friend (фамилия, адрес, номер телефона, дата рождения).

4. Создать базу (обобщенный список) из n записей, вывести полную информацию из базы на экран, отсортировав базу данных справочника по номеру телефона, а также организовать поиск в базе по фамилии.

Вариант 9

1. Создать абстрактный класс Client с функциями, позволяющими вывести на экран информацию о клиентах банка, а также определить соответствие клиента критерию поиска.
2. В абстрактном классе Client реализовать метод CompareTo так, чтобы можно было отсортировать базу данных о клиентах банка по дате открытия их счета.
3. Создать производные классы: Depositor (фамилия, дата открытия вклада, размер вклада, процент по вкладу), Credited (фамилия, дата выдачи кредита, размер кредита, процент по кредиту, остаток долга), Organization (название, дата открытия счета, номер счета, сумма на счету).
4. Создать базу (обобщенный список) из n клиентов, вывести полную информацию из базы на экран, отсортировав базу данных о клиентах банка по дате открытия их счета, а также организовать поиск клиентов, начавших сотрудничать с банком с заданной даты.

Вариант 10

1. Создать абстрактный класс Software с методами, позволяющими вывести на экран информацию о программном обеспечении, а также определить соответствие возможности использования (на момент текущей даты).
2. В абстрактном классе Software реализовать метод CompareTo так, чтобы можно было отсортировать базу данных по названию ПО.
3. Создать производные классы: FreeSoftware (название, производитель), SharewareSoftware (название, производитель, дата установки, срок бесплатного использования), ProprietarySoftware (название, производитель, цена, дата установки, срок использования).
4. Создать базу (обобщенный список) из n видов программного обеспечения, вывести полную информацию из базы на экран, отсортировав базу данных по названию ПО, а также организовать поиск программного обеспечения, которое допустимо использовать на текущую дату.

Вариант 11

1. Создать абстрактный класс Realty с функциями, позволяющими вывести на экран информацию об объектах недвижимости, а также определить соответствие записи критерию поиска.
2. В абстрактном классе Realty реализовать метод CompareTo так, чтобы можно было отсортировать базу данных недвижимости по стоимости.
3. Создать производные классы: Apartment (площадь, количество комнат, адрес, этаж, наличие балкона), House (площадь, количество комнат, адрес, этажность, наличие гаража), Garage (площадь, количество машиномест, адрес).
4. Создать базу (обобщенный список) из n записей, вывести полную информацию из базы на экран, отсортировав базу данных недвижимости по стоимости, а также организовать поиск в базе по адресу (можно использовать упрощенный адрес - только район).

Вариант 12

1. Создать абстрактный класс Ticket с функциями, позволяющими вывести на экран информацию о доступных билетах, а также определить соответствие записи критерию поиска.
2. В абстрактном классе Ticket реализовать метод CompareTo так, чтобы можно было отсортировать базу данных билетов по стоимости.

3. Создать производные классы: Train (класс обслуживания, номер вагона, номер места), Bus (номер места), Air (авиакомпания, класс обслуживания).
4. Создать базу (обобщенный список) из n записей, вывести полную информацию из базы на экран, отсортировав базу данных билетов по стоимости, а также организовать поиск в базе по месту и времени отправления.

Вариант 13
1. Создать абстрактный класс учебных дисциплин Sciences с функциями, позволяющими вывести на экран информацию о дисциплинах, а также определить соответствие записи критерию поиска.
2. В абстрактном классе Sciences реализовать метод CompareTo так, чтобы можно было отсортировать базу данных дисциплин по количеству часов.
3. Создать производные классы: Engineering (количество часов, форма контроля(зачёт\ дифф.зачёт\экзамен), преподаватель, необходимое для обучения оборудование и\или ПО), Human (количество часов, форма контроля(зачёт\дифф.зачёт\экзамен), преподаватель).
4. Создать базу (обобщенный список) из n записей, вывести полную информацию из базы на экран, отсортировав базу данных по преподавателям и по часам, а также организовать поиск в базе по преподавателям.

Вариант 14
1. Создать абстрактный класс Vehicle (транспортное средство) с функциями, позволяющими вывести на экран информацию о транспортных средствах, а также определить соответствие записи критерию поиска.
2. В абстрактном классе Vehicle реализовать метод CompareTo так, чтобы можно было отсортировать базу данных по цене.
3. Создать производные классы: Plane (авиакомпания, количество пассажиров, цена, год выпуска), Car (цена, год выпуска) и Ship (порт приписки, количество пассажиров, цена, год выпуска).
4. Создать базу (обобщенный список) из n транспортный средств, вывести полную информацию из базы на экран, отсортировав базу данных по году выпуска, а также организовать поиск в базе транспортных средств по заданной вместимости пассажиров.

Вариант 15
1. Создать абстрактный класс Client с функциями, позволяющими вывести на экран информацию о клиентах интернет-магазина, а также определить соответствие клиента критерию поиска.
2. В абстрактном классе Client реализовать метод CompareTo() так, чтобы можно было отсортировать базу данных о покупателях по сумме заказов.
3. Создать производные классы: Person (фамилия, дата заказа, сумма заказа, процент скидки), Organization (название, дата заказа, сумма заказа, процент скидки, контактное лицо). Реализовать в них интерфейс ICloneable.
4. Создать базу (обобщенный список) из n клиентов, вывести полную информацию из базы на экран, отсортировав базу данных о покупателях по сумме заказов, а также организовать поиск клиентов, начавших сотрудничать с интернет-магазином с заданной даты.

Вариант 16
1. Создать абстрактный класс Person с функциями, позволяющими вывести на экран информацию о клиентах интернет-сервиса по подбору репетиторов, а также определить соответствие клиента критерию поиска.

2. В абстрактном классе Person реализовать метод CompareTo() так, чтобы можно было отсортировать базу данных по ценам.
3. Создать производные классы: Student (ФИО, дисциплина, класс школы/курс университета, дата размещения заказа, желаемая цена за час занятия(не выше), срок обучения), Teacher (ФИО, дисциплина, дата регистрации в системе, цена за час(не ниже), возраст, стаж работы, место работы). Реализовать в них интерфейс ICloneable.
4. Создать базу (обобщенный список) из n клиентов, вывести полную информацию из базы на экран, отсортировав базу данных по ценам заказов, а также организовать поиск клиентов, начавших сотрудничать с интернет-сервисом с заданной даты.

Вариант 17
1. Создать абстрактный класс PostalItem с функциями, позволяющими вывести на экран информацию о почтовых отправлениях, а также определить соответствие записи критерию поиска.
2. В абстрактном классе PostalItem реализовать метод CompareTo() так, чтобы можно было отсортировать базу данных по цене.
3. Создать производные классы: Mail (отправитель, получатель, стоимость доставки), Parcel (отправитель, получатель, вес, стоимость доставки) и Package (отправитель, получатель, вес, габариты, стоимость доставки). Реализовать в них интерфейс ICloneable.
4. Создать базу (обобщенный список) из n почтовых отправлений, вывести полную информацию из базы на экран, отсортировав базу данных по стоимости, а также организовать поиск в базе по весу.

Вариант 18
1. Создать абстрактный класс Animal с функциями, позволяющими вывести на экран информацию о животных, а также определить соответствие записи критерию поиска.
2. В абстрактном классе Animal реализовать метод CompareTo() так, чтобы можно было отсортировать базу данных по весу животного.
3. Создать производные классы: Mammal, Predator, Herbivorous. Необходимо разработать поля, наследуемые от базового класса, и собственные поля производных классов. Реализовать в них интерфейс ICloneable.
4. Создать базу (обобщенный список) из n животных, вывести полную информацию из базы на экран, отсортировав базу данных по весу, а также организовать поиск в базе по способу питания.

Вариант 19
1. Создать абстрактный класс Persona с функциями, позволяющими вывести на экран информацию о персоне, а также определить ее возраст (на момент текущей даты).
2. В абстрактном классе Persona реализовать метод CompareTo() так, чтобы можно было отсортировать базу данных о персонах по дате рождения.
3. Создать производные классы: Worker, Administrator, Engineer. Необходимо разработать поля, наследуемые от базового класса, и собственные поля производных классов. Реализовать в них интерфейс ICloneable.
4. Создать базу (обобщенный список) из n персон, вывести полную информацию из базы на экран, отсортировав базу данных о персонах по дате рождения, а также организовать поиск персон, чей возраст попадает в заданный диапазон.

Вариант 20
1. Создать абстрактный класс State с функциями, позволяющими вывести на экран информацию о государстве, а также определить его возраст (на момент текущей даты).

2. В абстрактном классе State реализовать метод CompareTo() так, чтобы можно было отсортировать базу данных о государствах по численности населения.
3. Создать производные классы: Republic(численность населения, дата основания, президент), Kingdom(численность населения, дата основания, президент/премьер-министр, король/королева), Federation(численность населения, дата основания, президент, регионы). Необходимо разработать поля, наследуемые от базового класса, и собственные поля производных классов. Реализовать в них интерфейс ICloneable.
4. Создать базу (обобщенный список) из n государств, вывести полную информацию из базы на экран, отсортировав базу данных о государствах по численности населения, а также организовать поиск государств, численность населения которых попадает в заданный диапазон.

Задание 2

Реализовать класс хеш-таблиц Hash() для хранения элементов описанной в задании 1 иерархии классов.

```
class Hash: IEnumerable
    {
        protected ArrayList keys;
        protected List<MyClass> values;
        …
    }
```

Для этого:

1. Реализовать конструкторы:

public Hash() - предназначен для создания пустой коллекции.
public Hash (int capacity*)* - создает пустую коллекцию с начальной емкостью, заданной параметром capacity.
public Hash (Hash h) - служит для создания коллекции, которая инициализируется элементами и емкостью коллекции, заданной параметром h.

2. Реализовать:

a) свойство Count, позволяющее получить количество элементов в коллекции;
b) индексатор;
c) методы для добавления одного Add(object key, MyClass t) или нескольких элементов AddRange(Hash h) в коллекцию;
d) методы для удаления одного Remove(object key) или нескольких элементов из коллекции RemoveAll(Hash h);
e) метод для поиска элемента по значению Find(MyClass value);
f) метод для клонирования коллекции;
g) метод для поверхностного копирования;
h) очистку коллекции;
i) также реализовать интерфейсы IEnumerable и Ienumerator (реализовать итератор).

В методе main() :

а) Заполнить созданную коллекцию объектами. Используя меню, реализовать в программе добавление и удаление объектов коллекции.

b) Разработать и реализовать три запроса (количество элементов определенного вида, печать элементов определенного вида и т.п.).
c) Выполнить перебор элементов коллекции с помощью метода foreach.
d) Выполнить клонирование коллекции.
e) Выполнить поиск заданного элемента в коллекции.

Задание 3

1. Определить класс MyNewCollection, производный от класса Hash, который с помощью событий извещает об изменениях в коллекции. Коллекция изменяется:
   - при удалении/добавлении элементов
   - при изменении одной из входящих в коллекцию ссылок, например, когда одной из ссылок присваивается новое значение.

   В этом случае в соответствующих методах или свойствах класса бросаются события.
2. В новую версию класса MyNewCollection добавить
   - открытое автореализуемое свойство типа string с названием коллекции;
   - метод bool Remove (object key) для удаления элемента с ключом key ; если в списке нет элемента с таким ключом, метод возвращает значение false;
   - перегрузить индексатор, добавив выбрасывание события при изменении данных.

3. Для событий, извещающих об изменениях в коллекции, определяется свой делегат CollectionHandler с сигнатурой:

   void CollectionHandler (object source, CollectionHandlerEventArgs args);
4. Для передачи информации о событии определить класс CollectionHandlerEventArgs, производный от класса System.EventArgs, который содержит:
   - открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
   - открытое автореализуемое свойство типа string с информацией о типе изменений в коллекции;
   - открытое автореализуемое свойство для ссылки на объект, с которым связаны изменения;
   - конструкторы для инициализации класса;
   - перегруженную версию метода string ToString() для формирования строки с информацией обо всех полях класса.
5. В класс MyNewCollection добавить два события типа CollectionHandler:
   - CollectionCountChanged, которое происходит при добавлении нового элемента в коллекцию или при удалении элемента из коллекции; через объект CollectionHandlerEventArgs событие передает имя коллекции, строку с информацией о том, что в коллекцию был добавлен новый элемент или из нее был удален элемент, ссылку на добавленный или удаленный элемент;
   - CollectionReferenceChanged, которое происходит, когда одной из ссылок, входящих в коллекцию, присваивается новое значение; через объект CollectionHandlerEventArgs событие передает имя коллекции, строку с информацией о том, что был заменен элемент в коллекции, и ссылку на новый элемент.
6. Событие CollectionCountChanged бросают следующие методы класса MyNewCollection:
   - AddDefaults();
   - Add (object[]) ;
   - Remove (int index).
7. Событие CollectionReferenceChanged бросает метод set индексатора, определенного в классе MyNewCollection.

8. Информация об изменениях коллекции записывается в класс Journal, который хранит информацию в списке объектов типа JournalEntry. Каждый объект типа JournalEntry содержит информацию об отдельном изменении, которое произошло в коллекции. JournalEntry содержит:

- открытое автореализуемое свойство типа string с названием коллекции, в которой произошло событие;
- открытое автореализуемое свойство типа string с информацией о типе изменений в коллекции;
- открытое автореализуемое свойство типа string с данными объекта, с которым связаны изменения в коллекции;
- конструктор для инициализации полей класса;
- перегруженную версию метода string ToString().
- всех элементах массива.

9. Написать демонстрационную программу, в которой:

- создать две коллекции MyNewCollection.
- Создать два объекта типа Journal, один объект Journal подписать на события CollectionCountChanged и CollectionReferenceChanged из первой коллекции, другой объект Journal подписать на события CollectionReferenceChanged из обеих коллекций.

10. Внести изменения в коллекции MyNewCollection

- добавить элементы в коллекции;
- удалить некоторые элементы из коллекций;
- присвоить некоторым элементам коллекций новые значения.

11. Вывести данные обоих объектов Journal.