Приднестровский государственный университет им. Т.Г. Шевченко

физико-математический факультет

кафедра прикладной математики и информатики

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 6

по дисциплине:
«*Системы программирования*»

Тема:
«Наследование и полиморфизм»

РАЗРАБОТАЛИ:
ст. преподаватель кафедры ПМиИ
Великодный В.И.
ст. преподаватель кафедры ПМиИ
Калинкова Е.В.

г. Тирасполь

Цель работы:

Изучение приемов создания иерархии классов. Формирование представления о реализации принципа полиморфизма.

Теоретическая часть

Наследование

Основы наследования

Наследование является одним из трех основополагающих принципов объектно-ориентированного программирования, поскольку оно допускает создание иерархических классификаций. Благодаря наследованию можно создать общий класс, в котором определяются характерные особенности, присущие множеству связанных элементов. От этого класса могут затем наследовать другие, более конкретные классы, добавляя в него свои индивидуальные особенности.

В языке C# класс, который наследуется, называется ***базовым***, а класс, который наследует, – ***производным***. Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе, добавляя к ним свои собственные элементы.

Поддержка наследования в C# состоит в том, что в объявление одного класса разрешается вводить другой класс. Для этого при объявлении производного класса указывается базовый класс. При установке между классами отношения "является" строится зависимость между двумя или более типами классов. Базовая идея, лежащая в основе классического наследования, заключается в том, что новые классы могут создаваться с использованием существующих классов в качестве отправной точки.

Общая форма объявления класса, наследующего от базового класса:

```
class имя_производного_класса : имя_базового_класса
{
    // тело класса
}
```

Для любого производного класса можно указать только один базовый класс. В C# не предусмотрено наследование нескольких базовых классов в одном производном классе. (В этом отношении C# отличается от C++, где допускается наследование нескольких базовых классов. Данное обстоятельство следует принимать во внимание при переносе кода C++ в C#.) Тем не менее можно создать иерархию наследования, в которой производный класс становится базовым для другого производного класса. Разумеется, ни один из классов не может быть базовым для самого себя как непосредственно, так и косвенно. Но в любом случае производный класс наследует все члены своего базового класса, в том числе переменные экземпляра, методы, свойства и индексаторы.

Рассмотрим для начала простой пример. Ниже приведен класс `TwoDShape`, содержащий ширину и высоту двумерного объекта, например квадрата, прямоугольника, треугольника и т.д.

```
// Класс для двумерных объектов
class TwoDShape
{
    public double Width;
    public double Height;

    public void ShowDim()
    {
        Console.WriteLine("Ширина и высота равны {0} и {1}", Width, Height);
    }
}
```

Класс TwoDShape может стать базовым, т.е. отправной точкой для создания классов, описывающих конкретные типы двумерных объектов. Например, в приведенной ниже программе класс TwoDShape служит для порождения производного класса Triangle. Обратите особое внимание на объявление класса Triangle.

```
// Пример простой иерархии классов
using System;

// Класс для двумерных объектов
class TwoDShape
{
    public double Width;
    public double Height;

    public void ShowDim()
    {
        Console.WriteLine("Ширина и высота равны {0} и {1}", Width, Height);
    }
}

// Класс Triangle, производный от класса TwoDShape
class Triangle : TwoDShape
{
    public string Style; // тип треугольника

    // Вернуть площадь треугольника
    public double Area()
    {
        return Width * Height / 2;
    }

    // Показать тип треугольника
    public void ShowStyle()
    {
        Console.WriteLine("Треугольник {0}", Style);
    }
}

class Shapes
{
    static void Main()
    {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();

        t1.Width = 4.0;
        t1.Height = 4.0;
        t1.Style = "равнобедренный";

        t2.Width = 8.0;
        t2.Height = 12.0;
        t2.Style = "прямоугольный";

        Console.WriteLine("Сведения об объекте t1: ");
        t1.ShowStyle();
        t1.ShowDim();
        Console.WriteLine("Площадь равна {0}", t1.Area());

        Console.WriteLine();
```

```
        Console.WriteLine("Сведения об объекте t2: ");
        t2.ShowStyle();
        t2.ShowDim();
        Console.WriteLine("Площадь равна {0}", t2.Area());
    }
}
```

При выполнении этой программы получается следующий результат.

```
Сведения об объекте t1:
Треугольник равнобедренный
Ширина и высота равны 4 и 4
Площадь равна 8

Сведения об объекте t2:
Треугольник прямоугольный
Ширина и высота равны 8 и 12
Площадь равна 48
```

В классе `Triangle` создается особый тип объекта класса `TwoDShape` (в данном случае – треугольник). Кроме того, в класс `Triangle` входят все члены класса `TwoDShape`, к которым, в частности, добавляются методы `Area()` и `ShowStyle()`. Так, описание типа треугольника сохраняется в переменной `Style`, метод `Area()` рассчитывает и возвращает площадь треугольника, а метод `ShowStyle()` отображает тип треугольника.

В класс `Triangle` входят все члены его базового класса `TwoDShape`, и поэтому в нем переменные `Width` и `Height` доступны для метода `Area()`. Кроме того, объекты t1 и t2 в методе `Main()` могут обращаться непосредственно к переменным `Width` и `Height`, как будто они являются членами класса `Triangle`. На рис.1 схематически показано, каким образом класс `TwoDShape` вводится в класс `Triangle`.

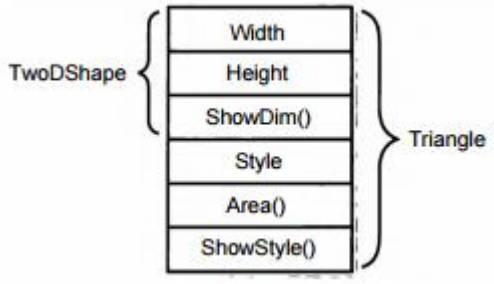


Рис. 1. Схематическое представление класса Triangle

Несмотря на то, что класс `TwoDShape` является базовым для класса `Triangle`, в то же время он представляет собой совершенно независимый и самодостаточный класс. Если класс служит базовым для производного класса, то это совсем не означает, что он не может быть использован самостоятельно. Например, следующий фрагмент кода считается вполне допустимым.

```
TwoDShape shape = new TwoDShape();
shape.Width = 10;
shape.Height = 20;
shape.ShowDim();
```

Разумеется, объект класса `TwoDShape` никак не связан с любым из классов, производных от класса `TwoDShape`, и вообще не имеет к ним доступа.

Главное преимущество наследования заключается в следующем: как только будет создан базовый класс, в котором определены общие для множества объектов атрибуты, он может быть использован для создания любого числа более конкретных производных классов. А в каждом производном классе может быть точно выстроена своя собственная классификация. В качестве примера ниже приведен еще один класс, производный от класса `TwoDShape` и инкапсулирующий прямоугольники.

```
// Класс для прямоугольников, производный от класса TwoDShape
class Rectangle : TwoDShape
```

```
{
    // Возвратить логическое значение true, если
    // прямоугольник является квадратом
    public bool IsSquare()
    {
        if (Width == Height) return true;
        return false;
    }
    // Возвратить площадь прямоугольника
    public double Area()
    {
        return Width * Height;
    }
}
```

В класс `Rectangle` входят все члены класса `TwoDShape`, к которым добавлен метод `IsSquare()`, определяющий, является ли прямоугольник квадратом, а также метод `Area()`, вычисляющий площадь прямоугольника.

Вызов конструкторов базового класса

В иерархии классов допускается, чтобы у базовых и производных классов были свои собственные конструкторы. В связи с этим возникает следующий вопрос: какой конструктор отвечает за построение объекта производного класса: конструктор базового класса, конструктор производного класса или же оба? На этот вопрос можно ответить так: конструктор базового класса конструирует базовую часть объекта, а конструктор производного класса – производную часть этого объекта. И в этом есть своя логика, поскольку базовому классу неизвестны и недоступны любые элементы производного класса, а значит, их конструирование должно происходить раздельно.

Если конструктор определен только в производном классе, то все происходит очень просто: конструируется объект производного класса, а базовая часть объекта автоматически конструируется его конструктором, используемым по умолчанию.

Когда конструкторы определяются как в базовом, так и в производном классе, процесс построения объекта несколько усложняется, поскольку должны выполняться конструкторы обоих классов. В данном случае приходится обращаться к ключевому слову `base`, которое находит двоякое применение: во-первых, для вызова конструктора базового класса; и во-вторых, для доступа к члену базового класса, скрывающегося за членом производного класса.

С помощью формы расширенного объявления конструктора производного класса и ключевого слова `base` в производном классе может быть вызван конструктор, определенный в его базовом классе. Ниже приведена общая форма этого расширенного объявления:

```
конструктор_производного_класса(список_параметров) : base(список_аргументов)
{
   // тело конструктора
}
```

где *список_аргументов* обозначает любые аргументы, необходимые конструктору в базовом классе. Обратите внимание на местоположение двоеточия.

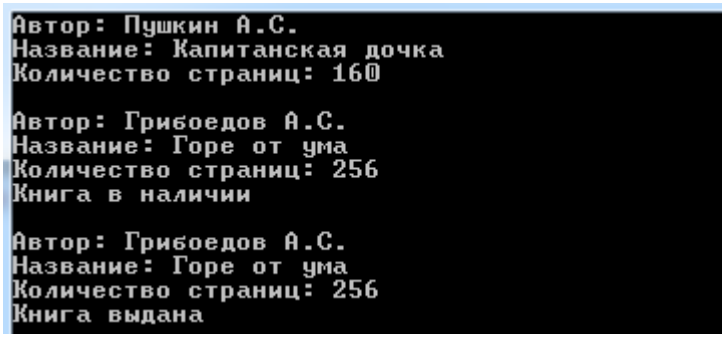
Пример.

```
using System;

// базовый класс
class Book
{
    // закрытые поля класса Book
    string Author;
```

```
    string Title;
    int Pages;

    // конструктор базового класса
    public Book(string author, string title, int pages)
    {
        Author = author;
        Title = title;
        Pages = pages;
    }

    // метод
    public void InfoBook()
    {
        Console.WriteLine("Автор: {0}\nНазвание: {1}\nКоличество страниц: {2}",
Author, Title, Pages);
    }
}

// производный класс, унаследованный от класса Book
class LibraryBook : Book
{
    // открытое поле класса LibraryBook
    public bool Status;  // равен true, если книга в наличии

    // конструктор производного класса
    // поля класса Book доступны через конструктор наследуемого класса
    public LibraryBook(string author, string title, int pages, bool status)
        : base(author, title, pages)
    {
        Status = status;
    }

    // метод
    public void IsInLibrary()
    {
        if (Status) Console.WriteLine("Книга в наличии");
        else Console.WriteLine("Книга выдана");
    }
}

class Program
{
    static void Main()
    {
        Book book1 = new Book("Пушкин А.С.", "Капитанская дочка", 160);
        LibraryBook book2 = new LibraryBook("Грибоедов А.С.", "Горе от ума", 256,
true);
        book1.InfoBook();
        Console.WriteLine();

        book2.InfoBook();
        book2.IsInLibrary();
        Console.WriteLine();

        book2.Status = false;
        book2.InfoBook();
        book2.IsInLibrary();
```

```
        Console.ReadLine();
    }
}
```

Результат работы программы:



С помощью ключевого слова `base` можно вызвать конструктор любой формы, определяемой в базовом классе, причем выполняться будет лишь тот конструктор, параметры которого соответствуют переданным аргументам.

А теперь рассмотрим вкратце основные принципы действия ключевого слова `base`. Когда в производном классе указывается ключевое слово `base`, вызывается конструктор из его непосредственного базового класса. Следовательно, ключевое слово `base` всегда обращается к базовому классу, стоящему в иерархии непосредственно над вызывающим классом. Это справедливо даже для многоуровневой иерархии классов. Аргументы передаются базовому конструктору в качестве аргументов метода `base()`. Если же ключевое слово отсутствует, то автоматически вызывается конструктор, используемый в базовом классе по умолчанию.

Полиморфизм и переопределение методов

Сам термин полиморфизм можно перевести как «*много форм*». А если говорить простыми словами, ***полиморфизм*** – это различная реализация однотипных действий. Классическая фраза, которая коротко объясняет полиморфизм – «Один интерфейс, множество реализаций».

Полиморфизм позволяет писать более абстрактные, расширяемые программы, один и тот же код используется для объектов разных классов, улучшается читабельность кода. Полиморфизм позволяет избавить разработчика от написания, чтения и отладки множества `if-else/switch-case` конструкций.

Виртуальный метод – это метод, который может быть переопределен в классе наследнике. При этом, при вызове будет вызываться метод не для класса, с которым объявлена переменная, а для реально хранящегося в переменной объекта.

Переопределение метода – это изменение его реализации в классе наследнике. Переопределив метод, он будет работать по-разному в базовом классе и классе наследнике, имея при этом одно и то же имя, аргументы и тип возврата.

Например, есть класс *Геометрическая Фигура*, и в нем объявлен метод `Draw()`, который будет рисовать фигуру. От этого класса наследуются классы Треугольник, Прямоугольник, Окружность. В них реализуется метод для рисования (понятно, что реализация рисования каждой фигуры разная). В итоге мы можем создать объекты этих классов, и у всех будет метод Draw(), который будет рисовать соответствующую фигуру.

Виртуальный метод объявляется при помощи ключевого слова `virtual`:

```
[модификатор доступа] virtual [тип] [имя метода] ([аргументы])
{
   // тело метода
}
```

Статический метод не может быть виртуальным.

Объявив виртуальный метод, мы теперь можем переопределить его в классе наследнике. Для этого используется ключевое слово override:

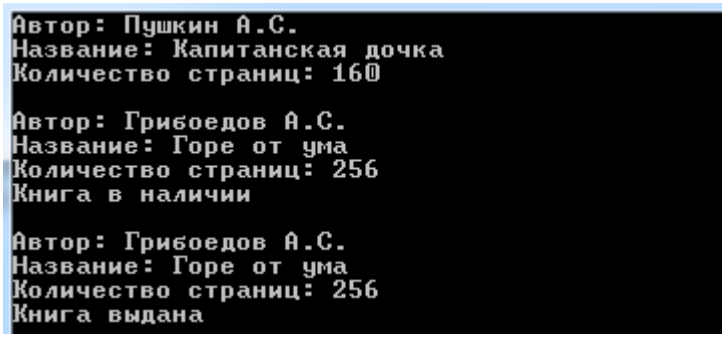
```
[модификатор доступа] override [тип] [имя метода] ([аргументы])
{
   // новое тело метода
}
```

Пример.

```
using System;

// базовый класс
class Book
{
    public string Author { get; set; }
    public string Title { get; set; }
    public int Pages { get; set; }

    public Book(string author, string title, int pages)
    {
        Author = author;
        Title = title;
        Pages = pages;
    }

    public virtual void InfoBook() // объявление виртуального метод
    {
        Console.WriteLine("Автор: {0}\nНазвание: {1}\nКоличество страниц: {2}",
Author, Title, Pages);
    }
}

// производный класс, унаследованный от класса Book
class LibraryBook : Book
{
    public bool Status { get; set; }

    public LibraryBook(string author, string title, int pages, bool status)
        : base(author, title, pages)
    {
        Status = status;
    }

    public override void InfoBook() // переопределение метода
    {
        Console.WriteLine("Автор: {0}\nНазвание: {1}\nКоличество страниц: {2}",
Author, Title, Pages);
        if (Status) Console.WriteLine("Книга в наличии");
        else Console.WriteLine("Книга выдана");
    }
}

class Program
{
    static void Main()
    {
        Book book1 = new Book("Пушкин А.С.", "Капитанская дочка", 160);
        LibraryBook book2 = new LibraryBook("Грибоедов А.С.", "Горе от ума", 256,
true);
```

```
        book1.InfoBook();
        Console.WriteLine();

        book2.InfoBook();
        Console.WriteLine();

        book2.Status = false;
        book2.InfoBook();

        Console.ReadLine();
    }
}
```

Результат работы программы:



Вызов базового метода

Бывает так, что функционал метода, который переопределяется, в базовом классе мало отличается от функционала, который должен быть определен в классе наследнике. В таком случае, при переопределении, мы можем вызвать сначала этот метод из базового класса, а дальше дописать необходимый функционал. Это делается при помощи ключевого слова `base`:

```
public virtual void InfoBook() // InfoBook в классе Book
{
    Console.WriteLine("Автор: {0}\nНазвание: {1}\nКоличество страниц: {2}",
Author, Title, Pages);
}
public override void InfoBook() // InfoBook в классе LibraryBook
{
    base.InfoBook();
    if (Status) Console.WriteLine("Книга в наличии");
    else Console.WriteLine("Книга выдана");
}
```

Абстрактные классы

Кроме обычных классов в C# есть абстрактные классы. Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. Но мы не можем создать объект или экземпляр абстрактного класса. Абстрактные классы лишь предоставляют базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

При определении абстрактных классов используется ключевое слово `abstract`:

```
abstract class Human
{
    public int Length { get; set; }
    public double Weight { get; set; }
}
```

Кроме обычных методов абстрактный класс может иметь абстрактные методы. Подобные методы определяются с помощью ключевого слова `abstract` и не имеют никакого функционала:

```
public abstract void Display();
```

При этом производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе. При переопределении в производном классе такой метод также объявляется с модификатором `override`. Также следует учесть, что если класс имеет хотя бы одно абстрактное свойство или метод, то он должен быть определен как абстрактный.

Абстрактные методы так же, как и виртуальные, являются частью полиморфного интерфейса. Но если в случае с виртуальными методами мы говорим, что класс-наследник наследует реализацию, то в случае с абстрактными методами наследуется интерфейс, представленный этими абстрактными методами.

Пример. Допустим, в нашей программе для банковского сектора мы можем определить три класса: `Person`, который описывает человека, `Employee`, который описывает сотрудника банка, и класс `Client`, который будет представлять клиента банка. Очевидно, что классы `Employee` и `Client` будут производными от класса `Person`. И так как все объекты будут представлять либо сотрудника банка, либо клиента, то напрямую мы от класса `Person` создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным.

```
using System;

// абстрактный класс, описывающий человека
abstract class Person
{
    // поля класса Person
    public string SurName; // фамилия
    public string Name;    // имя

    // конструктор
    public Person(string sName, string name)
    {
        SurName = sName;
        Name = name;
    }

    // абстрактный метод
    public abstract void Display();
}

// производный класс; описывает клиента банка
class Client : Person
{
    // поле класса Client
    public string Bank;   // название банка

    // конструктор
    public Client(string sName, string name, string comp)
        : base(sName, name)
    {
        Bank = comp;
    }

    // переопределенный метод
    public override void Display()
    {
        Console.WriteLine("{0} {1} имеет счет в банке {2}", Name, SurName, Bank);
```

```
    }
}

// производный класс; описывает сотрудника банка
class Employee : Person
{
    // поля класса Employee
    public string Bank;   // название банка
    public string Post;   // должность

    // конструктор
    public Employee(string sName, string name, string comp, string post)
        : base(sName, name)
    {
        Bank = comp;
        Post = post;
    }

    // переопределенный метод
    public override void Display()
    {
        Console.WriteLine("{0} {1} - {2} банка {3} ", Name, SurName, Post, Bank);
    }
}
class Program
{
    static void Main()
    {
        Client man1 = new Client("Петров","Михаил","Империал");
        man1.Display();
        Employee man2 = new Employee("Кузнецов","Илья","Ипотечный","бухгалтер");
        man2.Display();
        Console.ReadLine();
    }
}
```

Результат работы программы:

```
Михаил Петров имеет счет в банке Империал
Илья Кузнецов - бухгалтер банка Ипотечный
```

Интерфейсы

Иногда в объектно-ориентированном программировании полезно определить, что именно должен делать класс, но не как он должен это делать. Примером тому может служить упоминавшийся ранее абстрактный метод. В абстрактном методе определяются возвращаемый тип и сигнатура метода, но не предоставляется его реализация. А в производном классе должна быть обеспечена своя собственная реализация каждого абстрактного метода, определенного в его базовом классе. Таким образом, абстрактный метод определяет интерфейс, но не реализацию метода. Конечно, абстрактные классы и методы приносят известную пользу, но положенный в их основу принцип может быть развит далее. В C# предусмотрено разделение интерфейса класса и его реализации с помощью ключевого слова `interface`.

С точки зрения синтаксиса интерфейсы подобны абстрактным классам. Но в интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Как только интерфейс будет определен, он может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов.

Для реализации интерфейса в классе должны быть предоставлены тела (т.е. конкретные реализации) методов, описанных в этом интерфейсе. Каждому классу предоставляется полная свобода для определения деталей своей собственной реализации интерфейса. Следовательно, один и тот же интерфейс может быть реализован в двух классах по-разному. Тем не менее, в каждом из них должен поддерживаться один и тот же набор методов данного интерфейса. А в том коде, где известен такой интерфейс, могут использоваться объекты любого из этих двух классов, поскольку интерфейс для всех этих объектов остается одинаковым. Благодаря поддержке интерфейсов в C# может быть в полной мере реализован главный принцип полиморфизма: один интерфейс – множество методов.

Интерфейсы объявляются с помощью ключевого слова `interface`. Ниже приведена упрощенная форма объявления интерфейса.

```
interface имя
{
    возвращаемый_тип имя_метода1(список_параметров);
    возвращаемый_тип имя_метода2(список_параметров);
    // ...
    возвращаемый_тип имя_методаN(список_параметров);
}
```

где *имя* – это конкретное имя интерфейса. В объявлении методов интерфейса используются только их возвращаемый_тип и сигнатура. Они, по существу, являются абстрактными методами. Как пояснялось выше, в интерфейсе не может быть никакой реализации. Поэтому все методы интерфейса должны быть реализованы в каждом классе, включающем в себя этот интерфейс. В самом же интерфейсе методы неявно считаются открытыми, поэтому доступ к ним не нужно указывать явно.

Ниже приведен пример объявления интерфейса для класса, генерирующего последовательный ряд чисел.

```
public interface ISeries
{
    int GetNext(); // возвратить следующее по порядку число
    void Reset(); // перезапустить
    void SetStart(int x); // задать начальное значение
}
```

Этому интерфейсу присваивается имя `ISeries`. Префикс `I` в имени интерфейса указывать необязательно, но это принято делать в практике программирования, чтобы как-то отличать интерфейсы от классов. Интерфейс `ISeries` объявляется как `public` и поэтому может быть реализован в любом классе какой угодно программы.

Помимо методов, в интерфейсах можно также указывать свойства, индексаторы и события. Интерфейсы не могут содержать члены данных. В них нельзя также определить конструкторы, деструкторы или операторные методы. Кроме того, ни один из членов интерфейса не может быть объявлен как `static`.

Как только интерфейс будет определен, он может быть реализован в одном или не скольких классах. Для реализации интерфейса достаточно указать его имя после имени класса, аналогично базовому классу. Ниже приведена общая форма реализации интерфейса в классе.

```
class имя_класса : имя_интерфейса
{
    // тело класса
}
```

где *имя_интерфейса* – это конкретное имя реализуемого интерфейса. Если уж интерфейс реализуется в классе, то это должно быть сделано полностью. В частности, реализовать интерфейс выборочно и только по частям нельзя.

В классе допускается реализовывать несколько интерфейсов. В этом случае все реализуемые в классе интерфейсы указываются списком через запятую. В классе можно наследовать базовый класс и в тоже время реализовать один или более интерфейс. В таком случае имя базового класса должно быть указано перед списком интерфейсов, разделяемых запятой.

Методы, реализующие интерфейс, должны быть объявлены как `public`. Дело в том, что в самом интерфейсе эти методы неявно подразумеваются как открытые, поэтому их реализация также должна быть открытой. Кроме того, возвращаемый тип и сигнатура реализуемого метода должны точно соответствовать возвращаемому типу и сигнатуре, указанным в определении интерфейса.

В классах, реализующих интерфейсы, разрешается и часто практикуется определять их собственные дополнительные члены.

Ниже приведен **пример** программы, в которой реализуется представленный ранее интерфейс `ISeries`. В этой программе создается класс `ByTwos`, генерирующий последовательный ряд чисел, в котором каждое последующее число на два больше предыдущего.

```
// Реализовать интерфейс ISeries
class ByTwos : ISeries
{
    int start;
    int val;

    public ByTwos()
    {
        start = 0;
        val = 0;
    }

    public int GetNext()
    {
        val += 2;
        return val;
    }

    public void Reset()
    {
        val = start;
    }

    public void SetStart(int x)
    {
        start = x;
        val = start;
    }
}
```

В классе `ByTwos` реализуются три метода, определяемых в интерфейсе `ISeries`. Как пояснялось выше, это приходится делать потому, что в классе нельзя реализовать интерфейс частично.

Ниже приведен код класса, в котором демонстрируется применение класса `ByTwos`, реализующего интерфейс `ISeries`.

```
class SeriesDemo
{
    static void Main()
    {
        ByTwos ob = new ByTwos();
        for (int i = 0; i < 5; i++)
```

```
            Console.WriteLine("Следующее число равно " + ob.GetNext());
        Console.WriteLine("\nСбросить");
        ob.Reset();
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Следующее число равно " + ob.GetNext());
        Console.WriteLine("\nНачать с числа 100");
        ob.SetStart(100);
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Следующее число равно " + ob.GetNext());
        Console.ReadLine();
    }
}
```

Результат работы программы:

```
Следующее число равно 2
Следующее число равно 4
Следующее число равно 6
Следующее число равно 8
Следующее число равно 10

Сбросить
Следующее число равно 2
Следующее число равно 4
Следующее число равно 6
Следующее число равно 8
Следующее число равно 10

Начать с числа 100
Следующее число равно 102
Следующее число равно 104
Следующее число равно 106
Следующее число равно 108
Следующее число равно 110
```

Сравнение объектов

Для сравнения объектов в C# недостаточно просто определить операции ==, > и <, так как внутренние механизмы платформы .NET для сравнения используют не операции, а специальные методы. Сравнение можно разделить на проверку на равенство и собственно сравнение по величине.

Рассмотрим задачу проверки на равенство двух объектов. Эта задача встречается очень часто, поэтому методы для сравнения являются частью класса object (System.Object), а значит наследуются всеми классами. К этим методам относятся:

- Equals (их два: статический невиртуальный и виртуальный) — используется для сравнения объектов по значению. Например, если сравнивается площадь, то два различных объекта класса «Прямоугольник» могут оказаться равными.
- ReferenceEquals (статический) — проверяет на равенство ссылки. То есть, этот метод позволяет узнать, хранят ли две переменные ссылки на один и тот же объект или на два разных.
- GetHashCode (виртуальный) — возвращает так называемый хеш.

Хеш — это целое число, которое сопоставляется с каждым объектом и используется для ускорения сравнения. Для хеша выполняются свойства:

- если хеши различны, то объекты не равны,
- если хеши равны, то требуется дополнительная проверка.

Ускорение сравнения достигается за счёт того, что хеши обычно вычисляются быстрее, чем выполняется сравнение. То есть, если у двух объектов не совпали хеши, то можно их не сравнивать, они точно различны.

Пример хеша в реальной жизни — первая буква фамилии. Если нужно найти человека в большом списке, то можно сравнивать первые буквы, не читая всю фамилию. Если буквы не совпали, то фамилию пропускаем. Если совпали — сравниваем как обычно. На большом списке это быстрее, чем сравнивать фамилии целиком.

Хеш часто вычисляют как побитовое исключающее или (в C# эта операция обозначается знаком ^) от хешей полей или значений, важных для сравнения. Но единого рецепта нет, всё зависит от задачи.

Для того, чтобы реализовать проверку на равенство двух объектов обычно выполняют следующее:

- переопределяют (override) унаследованный виртуальный метод Equals, принимающий object,
- реализуют интерфейс IEquatable<T> — в нём тоже определён метод Equals, но с параметром конкретного типа (можно реализовать несколько вариантов этого интерфейса для сравнения с разными типами),
- переопределяют (override) метод GetHashCode,
- определяют операции == и !=.

Не все из этих пунктов обязательны, но для корректной и удобно работы с объектами лучше выполнить их все.

При сравнении нужно учесть, что второй объект может иметь другой тип или может равняться null. Кроме того, должны выполняться математические свойства равенства:

- $x = x$,
- $x = y \Rightarrow y = x$,
- $x = y \wedge y = z \Rightarrow x = z$.

Рассмотрим пример реализации проверки на равенство. Опишем класс Rectangle (прямоугольник с целочисленными шириной и высотой). Будем сравнивать прямоугольники по площади.

```
using System;

// Интерфейс IEquatable<Rectangle> говорит о том,
// что наш объект можно сравнивать с другим объектом
// типа Rectangle
class Rectangle : IEquatable<Rectangle>
{
    // Ширина и высота
    public int Width { get; set; }
    public int Height { get; set; }

    // Переопределение унаследованного метода Equals
    public override bool Equals(object obj)
    {
        // Чтобы не дублировать код, используем
        // реализацию Equals для интерфейса

        // Приводим obj к типу Rectangle
        return Equals(obj as Rectangle);
    }

    // Реализация метода из интерфейса IEquatable<Rectangle>
    public bool Equals(Rectangle rect)
    {
        // Ни один объект не равен null
        // (Приведение к object нужно, чтобы была вызвана
        // реализация == для object, а не для нашего класса,
        // так как в нашем классе == вызывает этот метод и
        // будет циклический вызов)
        if ((object)rect == null)
            return false;

        // Дополнительно проверим, что объект действительно
```

```
        // прямоугольник, а не потомок класса Rectangle
        // (такая проверка не всегда требуется, это зависит
        // от задачи)
        if (rect.GetType() != typeof(Rectangle))
            return false;

        // Любой объект равен самому себе
        if (object.ReferenceEquals(this, rect))
            return false;

        // Считаем, что два прямоугольника равны, если равны
        // их площади.
        long S1 = rect.Width * rect.Height;
        long S2 = Width * Height;
        return S1 == S2;
    }

    public override int GetHashCode()
    {
        // Вычислим хеш прямоугольника как хеш его площади
        long S = Width * Height;
        return S.GetHashCode();
    }

    // Определяем операции сравнения
    public static bool operator ==(Rectangle r1, Rectangle r2)
    {
        return r1.Equals(r2);
    }

    public static bool operator !=(Rectangle r1, Rectangle r2)
    {
        return !r1.Equals(r2);
    }
}

class Program
{
    static void Main()
    {
        Rectangle r1 = new Rectangle() { Width = 2, Height = 3 };
        Rectangle r2 = new Rectangle() { Width = 3, Height = 2 };
        Rectangle r3 = new Rectangle() { Width = 1, Height = 1 };

        Console.WriteLine(r1 == r2);        // True
        Console.WriteLine(r1 == r3);        // False
        Console.WriteLine(r1.Equals(r2));   // True
        Console.WriteLine(r1.Equals(r3));   // False
    }
}
```

Если убрать комментарии, реализация получается достаточно компактной. Её можно ещё немного упростить, если использовать сложные условия, но программу станет сложнее читать.

Для реализации сравнения необходимо реализовать интерфейс IComparable<T> и определить операции сравнения.

Интерфейс содержит единственный метод CompareTo, возвращающий целое число, которое:

- больше нуля, если первый объект больше второго,

- равно нулю, если объекты равны,
- меньше нуля, если первый объект меньше второго.

При реализации нужно соблюдать математические свойства операций сравнения. Кроме того, обычно считают, что null меньше любого объекта.

Пример реализации:

```
using System;

class Rectangle : IComparable<Rectangle>
{
    // Ширина и высота
    public int Width { get; set; }
    public int Height { get; set; }

    public int CompareTo(Rectangle rect)
    {
        // Считаем, что любой объект больше null
        if ((object)rect == null)
            return 1;

        // Будем сравнивать площади
        long S1 = Width * Height;
        long S2 = rect.Width * rect.Height;

        // При сравнении чисел можно просто проверить знак
        // их разности, это позволит не писать if
        return Math.Sign(S1 - S2);
    }

    public static bool operator > (Rectangle r1, Rectangle r2)
    {
        return r1.CompareTo(r2) > 0;
    }

    public static bool operator < (Rectangle r1, Rectangle r2)
    {
        return r1.CompareTo(r2) < 0;
    }

    public static bool operator >= (Rectangle r1, Rectangle r2)
    {
        return r1.CompareTo(r2) >= 0;
    }

    public static bool operator <= (Rectangle r1, Rectangle r2)
    {
        return r1.CompareTo(r2) <= 0;
    }
}

class Program
{
    static void Main()
    {
        Rectangle r1 = new Rectangle() { Width = 2, Height = 3 };
        Rectangle r2 = new Rectangle() { Width = 3, Height = 2 };
        Rectangle r3 = new Rectangle() { Width = 1, Height = 1 };
```

```
        Console.WriteLine(r1 > r2);         // False
        Console.WriteLine(r1 >= r3);        // True
    }
}
```

Практическая часть

Рассмотрим несколько примеров.

Пример 1. Рассмотрим систему геометрических фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами:

```
using System;

// абстрактный класс фигуры
abstract class Shape
{
    double x; // x-координата точки
    double y; // y-координата точки

    // конструктор
    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    // абстрактный метод для получения периметра
    public abstract double Perimeter();

    // абстрактный метод для получения площади
    public abstract double Area();
}

// производный класс прямоугольника
class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    // конструктор с обращением к конструктору класса Shape
    public Rectangle(double x, double y, double width, double height)
        : base(x, y)
    {
        Width = width;
        Height = height;
    }

    // переопределение получения периметра
    public override double Perimeter()
    {
        return Width * 2 + Height * 2;
    }

    // переопределение получения площади
    public override double Area()
    {
        return Width * Height;
```

```
    }
}

class Program
{
    static void Main()
    {
        Shape rect1 = new Rectangle(2, 3, 7, 5);
        Console.WriteLine("Периметр прямоугольника равен {0}\nПлощадь
прямоугольника равна {1}", rect1.Perimeter(), rect1.Area());
    }
}
```

Результат работы программы:

```
Периметр прямоугольника равен 24
Площадь прямоугольника равна 35
```

Пример 2. Решить предыдущую задачу с использованием интерфейсов.

```
using System;

// интерфейс
interface IShape
{
    // методы
    double Area();
    void PrintInfo();
}

// данный класс реализует интерфейс IShape
class Triangle : IShape
{
    // поля класса
    double a;
    double b;
    double c;

    // конструктор
    public Triangle(double a, double b, double c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // реализация методов интерфейса
    public double Area()
    {
        double p = (a + b + c) / 2;
        return Math.Sqrt(p * (p - a) * (p - b) * (p - c));
    }

    public void PrintInfo()
    {
        Console.WriteLine("Площадь треугольника со сторонами {0}, {1}, {2} равна
{3} ", a, b, c, Area());
    }
}
```

```
// данный класс реализует интерфейс IShape
class Rectangle : IShape
{
    // поля класса
    double width;
    double height;

    // конструктор
    public Rectangle(double width, double height)
    {
        this.width = width;
        this.height = height;
    }

    // реализация методов интерфейса
    public double Area()
    {
        return width * height;
    }

    public void PrintInfo()
    {
        Console.WriteLine("Площадь прямоугольника со сторонами {0} и {1} равна
{2}", width, height, Area());
    }
}

class Program
{
    static void Main()
    {
        Triangle shape1 = new Triangle(3, 4, 5);
        Rectangle shape2 = new Rectangle(6, 7);
        shape1.PrintInfo();
        shape2.PrintInfo();
        Console.ReadLine();
    }
}
```

Результат работы программы:

```
Площадь треугольника со сторонами 3, 4, 5 равна 6
Площадь прямоугольника со сторонами 6 и 7 равна 42
```

Пример 3. Описать интерфейс для фигур с площадью. Описать классы треугольников и прямоугольников. Определить статический метод (функцию) для сравнения произвольных фигур по площади.

```
using System;

interface IShape // Интерфейс "Фигура"
{
    double Area();
}

// Треугольник
class Triangle : IShape
{
    // Стороны
```

```
    double a;
    double b;
    double c;

    public Triangle(double a, double b, double c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // реализация интерфейса
    public double Area()
    {
        double p = (a + b + c) / 2;
        return Math.Sqrt(p * (p - a) * (p - b) * (p - c));
    }

    public override string ToString()
    {
        return string.Format("Треугольник со сторонами {0}, {1}, {2}; площадь -
{3} ", a, b, c, Area());
    }
}

// Прямоугольник
class Rectangle : IShape
{
    // Ширина и высота
    double width;
    double height;

    public Rectangle(double width, double height)
    {
        this.width = width;
        this.height = height;
    }

    // реализация интерфейса
    public double Area()
    {
        return width * height;
    }

    public override string ToString()
    {
        return string.Format("Прямоугольник со сторонами {0} и {1}; площадь -
{2}", width, height, Area());
    }
}

class Program
{
    // Проверяет, что площадь первой фигуры строго больше
    static bool IsGreater(IShape s1, IShape s2)
    {
        return s1.Area() > s2.Area();
    }
```

```
    static void Main()
    {
        Triangle shape1 = new Triangle(3, 4, 5);
        Rectangle shape2 = new Rectangle(6, 7);

        Console.WriteLine(shape1);
        Console.WriteLine(shape2);

        if (IsGreater(shape1, shape2))
            Console.WriteLine("[{0}] больше [{1}]", shape1, shape2);
        else
            Console.WriteLine("[{0}] меньше либо равен [{1}]", shape1, shape2);
    }
}
```

Задания для самостоятельной работы

Разработайте приложения для решения задач из сборника задач по программированию согласно вашему варианту.