КЛАССЫ И ОБЪЕКТЫ

Основные понятия

Класс – это обобщенное понятие, определяющее характеристики и поведение некоторого множества объектов, называемых экземплярами класса. В программном понимании класс является типом данных, определяемым пользователем.

Описание класса содержит ключевое слово class, за которым следует его *имя*, а далее в фигурных скобках – *тело* класса. Кроме того, для класса можно задать его базовый класс (предок) и ряд необязательных атрибутов и спецификаторов, определяющих различные характеристики класса:

```
[ атрибуты ] [ спецификаторы ] class имя_класса [ : предки ]
{
   тело_класса
}
```

Замечание

.NET позволяет указывать только один класс в качестве предка. Кроме этого, класс может реализовывать несколько интерфейсов. Более подробно схема наследования будет рассмотрена позже.

Простейший пример класса:

```
class Demo
{
}
```

Спецификаторы определяют характеристики класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов перечислены в следующей таблице:

№	Спецификатор	Описание
1	abstract	Абстрактный класс. Применяется в иерархии объектов.
2	internal	Доступ только из данного проекта (сборки).
3	new	Задает новое описание класса взамен унаследованного от предка. Используется для вложения классов (в иерархии объектов).
4	private	Доступ только из элементов класса, внутри которых описан данный класс. Используется для вложенных классов.
5	protected	Доступ только из данного, или производного класса. Используется для вложенных классов.
6	protected internal	Доступ только из данного и производного класса в рамках текущего проекта (сборки).
7	public	Доступ к классу не ограничен
8	sealed	Бесплодный класс. Запрещает наследование данного класса. Применяется в иерархии объектов.

№	Спецификатор	Описание
9	static	Статический класс. Позволяет обращатся к методам класса без создания экземпляра класса.

Спецификаторы 2, 4-7 называются спецификаторами *доступа*. Они определяют, и заках мест программы можно непосредственно обращаться к данному классу. Спецификаторы доступа могут комбинироваться с остальными спецификаторами, но не могут комбинироваться между собой.

Замечание

В рамках данного курса атрибуты класса мы рассматривать не будем.

Класс можно описывать непосредственно внутри пространства имен, или внутри другого класса. В последнем случае класс называется вложенным. В зависимости от места описания класса, некоторые из этих спецификаторов могут быть запрещены. Например, в рамках данного курса мы будем рассматривать только классы, которые описываются непосредственно в пространстве имен (то есть, не являющиеся вложенными). Для таких классов допускаются только два спецификатора: public и internal. Если ни один спецификатор доступа не указан, то по умолчанию используется спецификатор internal.

Объекты (экземпляры класса) создаются явным образом с помощью операции new, например:

```
Demo a = new Demo (); // Создается экземпляр класса Demo
```

Если достаточный для хранения объекта объем памяти выделить не удалось, то генерируется исключение OutOfMemoryException.

Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его члены: данные и методы. В классе могут присутствовать статические члены, которые существуют в единственном экземпляре для всех объектов класса. Статические данные часто называют данными класса, а остальные – данными экземпляра. Для работы с данными класса используются статические методы класса, а для работы с данными экземпляра – методы экземпляра, или просто методы.

В общем случае класс может содержать следующие функциональные члены:

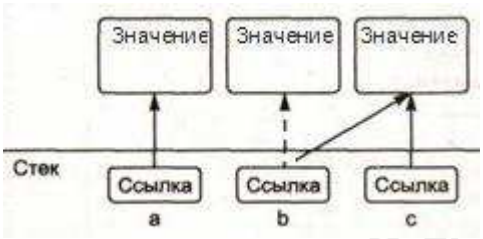
1) данные: переменные, или константы;
2) методы: реализуют не только вычисления, но и другие действия с классом, или его экземпляром;
3) конструкторы: реализуют действия по инициализации экземпляров класса, или его статических полей;
4) деструкторы: определяют действия, которые необходимо выполнить непосредственно перед уничтожением объекта;
5) свойства: определяют возможности доступа к членам класса;
6) индексаторы: обеспечивают возможность доступа к членам класса по их порядковому номеру (индексу);
7) операции: задают действия с экземплярами класса с помощью знаков операций;
8) события: определяют уведомления, которые может генерировать класс;
9) типы и структуры данных, определенные внутри класса.

Прежде чем приступить к проектированию классов, отметим, что классы относятся к ссылочным типам данных. Принципиальное различие между размерными и ссылочными типами состоит в способе хранения их значений в памяти. Для размерных типов фактическое

значение хранится в стеке (или как часть большого объекта ссылочного типа). Адрес переменной ссылочного типа тоже хранится в стеке, но сам объект хранится в куче (динамической памяти).

Данное различие существенно скажется на выполнении операций присваивания и сравнения объектов. Сам механизм выполнения присваивания один и тот же для величин любого типа, как ссылочного, так и размерного, однако результаты различаются. При присваивании значения копируется значение, а при присваивании ссылки – ссылка.

Например, пусть были созданы три объекта a, b и c, а затем выполнено присваивание b = c. Теперь ссылки b и c указывают на один и тот же объект. Старое значение b становится недоступным и удаляется сборщиком мусора.



Рассмотрим теперь операцию сравнения. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одну и ту же область памяти. Так, объекты b и c равны, т.к. они ссылаются на одну и ту же область памяти, но a не равно b даже при равенстве их значений.

Замечание

Исключение из этого правила составляют строки, которые являются ссылочными типами данных, но для которых операции сравнения сравнивают хранящиеся в динамической памяти значения, а не адреса ссылок. Если мы захотим получить аналогичный эффект для разработанного нами типа, нам придется перегрузить для него метод Equals и соответсвующие операторы.

Члены-данные: поля и константы

Члены-данные класса могут быть *переменными* или *константами* и должны задаваться в соответствии с правилами объявления идентификаторов.

Синтаксис описания членов-данных:

```
[атрибуты] [спецификаторы] [const] тип имя [ = начальное_значение];
```

Рассмотрим возможные спецификаторы для данных:

№	Спецификатор	Описание
1	internal	Доступ только из данной сборки.
2	new	Новое описание поля, скрывающее унаследованный элемент класса.
3	private	Доступ только из данного класса.
4	protected	Доступ только из данного и производных классов.
5	protected internal	Доступ только из данного и производных классов из данной сборки.
6	public	Доступ к элементу не ограничен.

№	Спецификатор	Описание
7	readonly	Поле доступно только для чтения (значения таких полей можно установить либо при описании, либо в конструкторе).
8	static	Одно поле для всех экземпляров класса.
9	volatile	Поле может изменяться другим процессом, или системой.

Для констант можно использовать только спецификаторы 1-6.

По умолчанию члены-данные считаются закрытыми, т.е., для них установлен спецификатор private. Для членов-данных этот вид доступа является предпочтительным, поскольку поля определяют внутреннее строение класса, которое должно быть скрыто от пользователя. Все методы класса имеют непосредственный доступ к его закрытым полям.

Поля описанные со спецификатором static, а также константы, существуют в единственном экземпляре для всех объектов класса, поэтому к ним обращаются не через имя экземпляра, а через имя класса. Обращение к полю класса выполняется с помощью операции доступа (точка). Справа от точки задается имя поля, слева – имя экземпляра для обычных полей и имя класса для статических. Рассмотрим пример создания класса Circle и возможные способы обращения к его полям.

Замечание

Пока на учебном примере мы будем размещать два класса в одном файле, а затем каждый класс будем помещен в собственный файл. Кроме этого, практически не будут использоваться документационные комментарии.

```
using System;
namespace MyProgram
{
   class Circle
   {
      public int x = 0;
      public int y = 0;
      public int radius = 3;
      public const double pi = 3.14;
      public static readonly string name = "Окружность";
      double area;
   }

   class Program
   {
      static void Main()
      {
         //создание экземпляра класса
         Circle oneCircle = new Circle();


         // обращение к константе
         Console.WriteLine("pi={0}", Circle.pi);

         // обращение к статическому полю
         Console.WriteLine("Используется объект {0}", Circle.name);
```

```
            //обращение к обычным полям
            Console.WriteLine("Центр в точке ({0},{1}), радиус {2}",
                        oneCircle.x, oneCircle.y, oneCircle.radius);
            oneCircle.radius = 100;
            Console.WriteLine(" Новая окружность с центром в точке
                        ({0},{1}) и радиусом {2}",
                        oneCircle.x, oneCircle.y, oneCircle.radius);

            //обращение к полю area вызовет ошибку, т.к. это
            //поле по умолчанию имеет спецификатор private
            //oneCircle.area = 2 * Circle.pi * oneCircle.radius;

            //попытка изменить значение поля name вызовет ошибку,
            // т.к. это поле доступно только для чтения
            //Circle.name="квадрат";
         }
      }
   }
```

Результат работы программы:

```
Используется объект Окружность
Центр в точке (0, 0), радиус 3
Новая окружность с центром в точке (0, 0) и радиусом 100
```

Задание

Уберите комментарии в строках, в которых происходит обращение к полям area и name, и посмотрите, какие сообщения выдаст компилятор.

Методы класса

Замечание

Создание и использование методов было рассмотрено нами ранее. Теперь рассмотрим использование методов в контексте создания классов.

Методы класса находятся в памяти в единственном экземпляре, и используются всеми объектами одного класса совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший метод экземпляр.

В явном виде параметр this применяется для того, чтобы возвратить из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например:

```
using System;
namespace MyProgram
{
   class Circle
   {
      public int x = 0;
      public int y = 0;
      public int radius = 3;
```

```
        public const double pi = 3.14;
        public static readonly string name = "Окружность";
        public void Set (int x, int y, int r)
        {
           // использует параметр this для обращения к полям
           // класса, т.к. их имена совпадают с
           // именами параметров метода
           this.x = x;
           this.y = y;
           radius=r;
        }
        public void Show()
        {
           Console.WriteLine("{0} с центром  в точке ({1},{2})
                              радиусом {3}", name, x, y, radius);
        }
     }
     class Program
     {
        static void Main()
        {
           Circle oneCircle = new Circle();
           oneCircle.Show();
           oneCircle.Set(1, 1, 100);
           oneCircle.Show();
        }
     }
  }
```

Результат работы программы:

```
Окружность с центром  в точке (0, 0) радиусом 3
Окружность с центром  в точке (1, 1) радиусом 100
```

Замечание

Здесь и далее метод Show мы будем использовать для вывода информации об объекте в консольное окно. Однако в общем случае в теле сущностного класса не должно быть методов, зависящих от выбранного интерфейса – консольного или оконного. У нас должна быть возможность использовать этот класс из приложений различного типа.

Задание

Добавьте в класс методы, вычисляющие

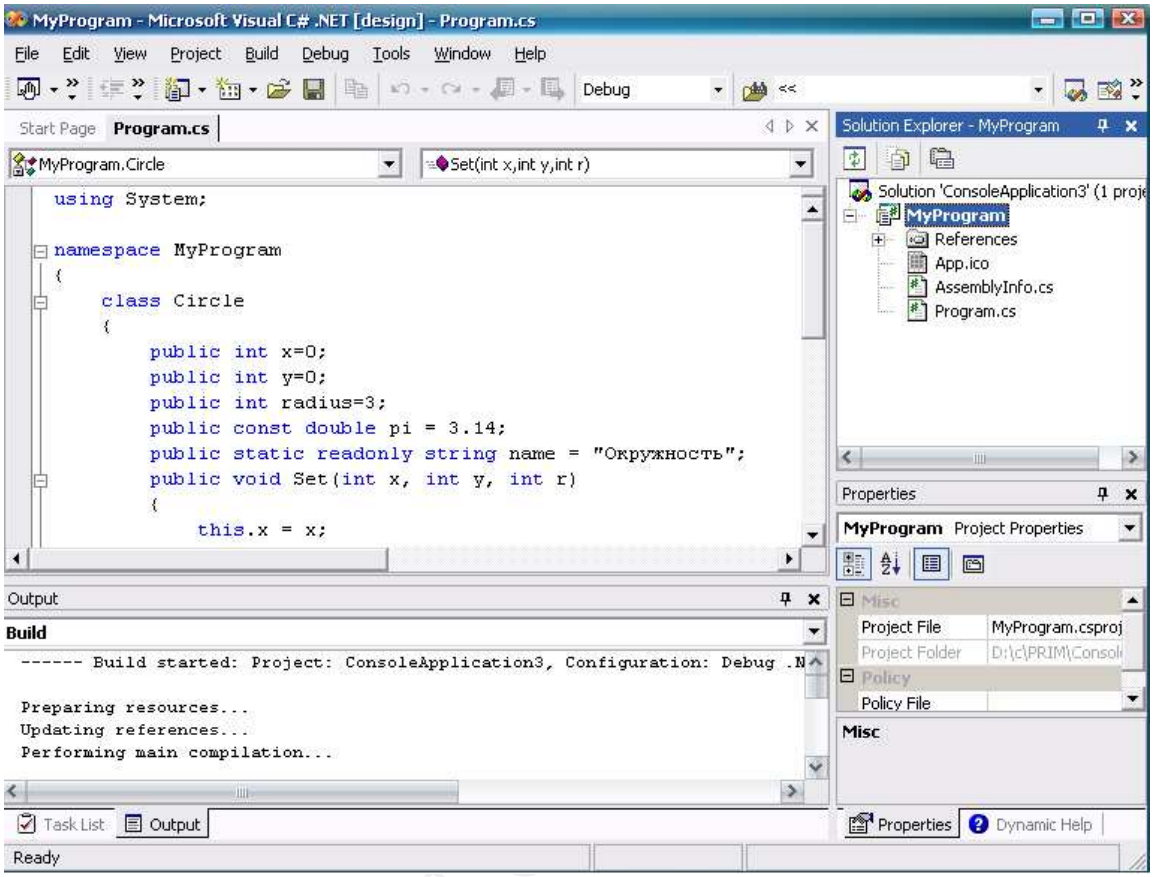
- *площадь круга;*
- *длину окружности*

и продемонстрируйте работу данных методов на примерах.

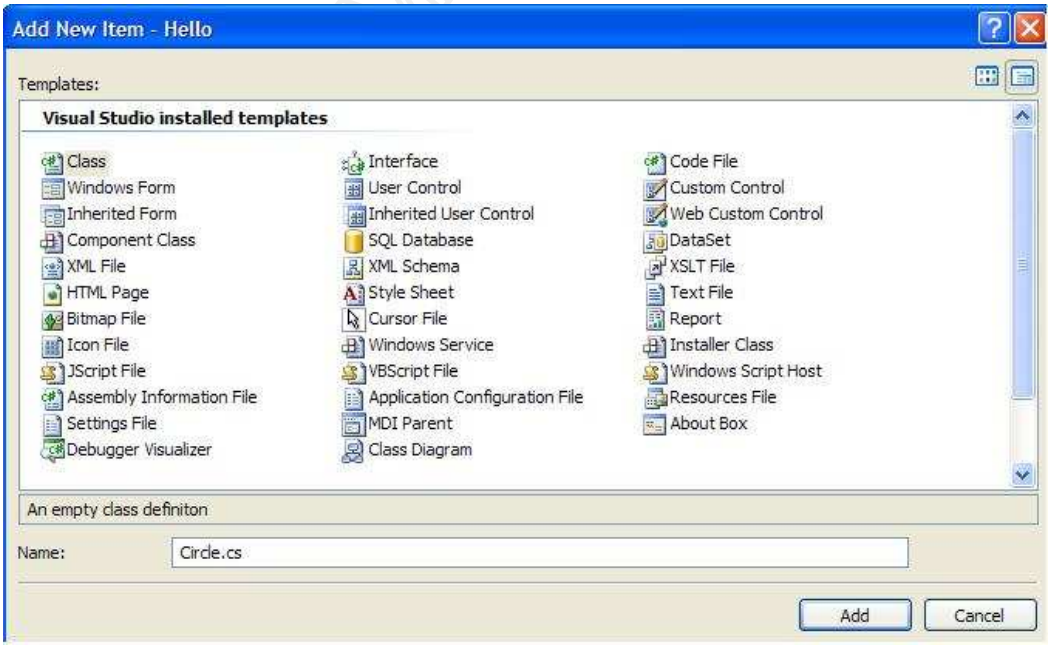
«Один класс – один файл»

С добавлением новых классов в программу резко увеличивается ее размер, что затрудняет ее прочтение. Поэтому следует руководствоваться одним простым принципом «один класс – один файл». Для того, чтобы создать новый файл для класса Circle выполним следующие действия:

1. В окне Solution Explorer щелкните правой кнопкой на имени проекта (В нашем случае проект называется MyProgram и его имя выделено).

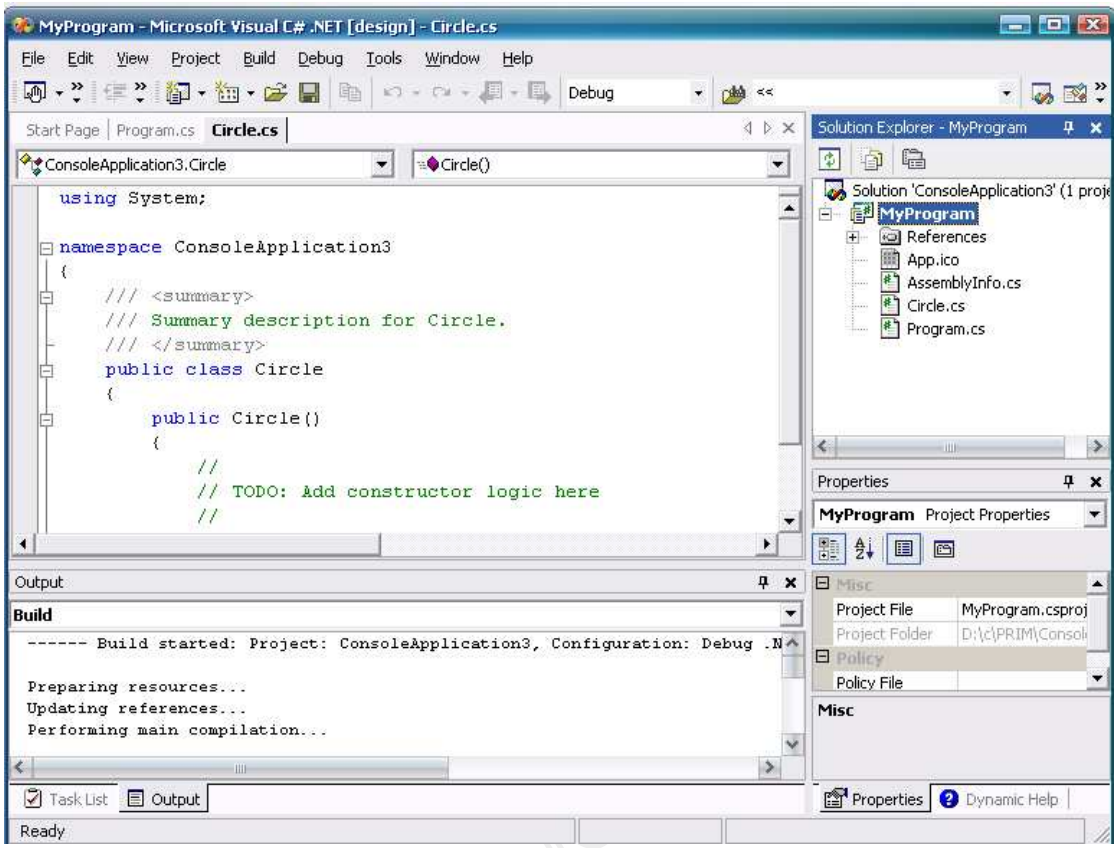


2. Выполните команду Add/Add Class…



В поле Name напишите Circle.cs и нажмите кнопку Add.

Теперь окно Solution Explorer выглядит следующим образом:



3. Замените namespace ConsoleApplication3 на namespace MyProgram, для того чтобы идентификаторы файлов Program.cs и Circle.cs были определены в одном пространстве имен.
4. Перенесите класс Circle из файла Program.cs в файл Circle.cs.
5. Теперь запустите программу, и посмотрите как она работает.

Конструкторы

Конструктор предназначен для инициализации объекта. Конструкторы делятся на конструкторы класса (для статических классов) и конструкторы экземпляра класса (для всех объектов класса).

Конструкторы экземпляра

Конструктор экземпляра вызывается автоматически при создании объекта класса с помощью операции new. Имя конструктора должно совпадать с именем класса.

Рассмотрим основные свойства конструкторов.

1. Конструктор не возвращает значение, даже типа void.
2. Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации. Выбор используемого конструктора происходит автоматически, в зависимости от количества и типа передаваемых параметров.
3. Если программист не указал ни одного конструктора, или какие-то поля не были инициализированы, то полям значимых типов присваивается нуль, полям ссылочных типов – null.

До сих пор мы задавали начальные значения полей класса при его описании. Это удобно в том случае, когда для всех экземпляров класса начальные значения полей одинаковы. Если же при создании объектов требуется присваивать полю разные значения, это следует делать с помощью конструктора.

Добавим в класс Circle два конструктора: первый из которох инициализирует только поле radius, а второй – поля x, y, radius.

```
using System;
namespace MyProgram
{
   class Circle
   {
      public int x;
      public int y;
      public int radius;
      public static readonly string name = "Окружность";
      //конструктор 1- инициализирует только поле radius
      public Circle( int r)
      {
         radius = r;
      }
      // конструктор 2 - инициализирует поля x, y, radius
      public Circle (int x, int y, int r)
      {
         this.x = x;
         this.y = y;
         radius = r;
      }
      public void Set(int x, int y, int r)
      {
         this.x = x;
         this.y = y;
         radius = r;
      }
      public void Show()
      {
         Console.WriteLine("{0} с центром  в точке ({1},{2}),
                           радиусом {3}", name, x, y, radius);
      }
   }
}
```

Если один из конструкторов выполняет какие-либо действия, а другой должен делать то же самое плюс еще что-то, то можно воспользоваться параметром this для того, чтобы вызвать один конструктор из другого. Например:

```
public Circle( int r) //конструктор 1
{
   radius=r;
}
```

```
public Circle (int x, int y, int r)
:this (r)  //конструктор 2
{
   this.x = x;
   this.y = y;
}
```

В данном случае конструктор 2 вызывает конструктор 1. Запись вида :this(r) называется инициализатором, то есть кодом, который будет выполнен до начала выполнения тела конструктора 2. Рассмотрим, как вызвать конструкторы класса Circle из класса Program:

```
using System;
namespace MyProgram
{
   class Program
   {
      static void Main()
      {
         //вызов конструктора 1
         Circle oneCircle = new Circle(1);
         oneCircle.Show();

         //вызов конструктора 2
         Circle twoCircle = new Circle(1, 1, 100);
         twoCircle.Show();
      }
   }
}
```

Результат работы программы:

```
Окружность с центром  в точке (0, 0) радиусом 1
Окружность с центром  в точке (1, 1) радиусом 100
```

Задание

Объясните:

1. *почему у первой окружности координаты x и y приняли значения 0, 0, хотя они и не были явным образом указаны при вызове конструктора;*
2. *какое сообщение будет выдано компилятором при выполнении команды:*

   ```
   Circle oneCircle = new Circle();
   ```

   и почему?

Замечание

Многие механизмы .NET, предназначенные для сериализации объектов, требуют наличие у класса конструктора по-умолчанию. В рамках данного пособия эти механизмы рассматриваться не будут, но стоит помнить о том, что удалять конструктор по-умолчанию без экстренной необходимости не рекомандуется.

Конструкторы класса

Статические классы содержат только статические члены (в том числе, и конструктор), которые хранятся в памяти в единственном экземпляре. Поэтому создавать экземпляры класса для них нет смысла.

Чтобы подчеркнуть этот факт, в первой версии C# для статических классов создавали два конструктора, один – пустой закрытый (private) конструктор, второй – статический конструктор, не имеющий параметров. Первый конструктор предотвращал попытки создания экземпляров класса. Второй конструктор автоматически вызывался системой до первого обращения к любому элементу статического класса, выполняя необходимые действия по инициализации. Вышесказанное отражено в следующем примере:

```
class Demo
{
   static int a;
   static int b;
   private Demo(){}    //закрытый конструктор
   static Demo()       //статический конструктор
   {
      a = 10;
      b = 2;
   }
   public static void Print ()
   {
      Console.WriteLine("{0}+{1}={2}", a, b, a+b);
      Console.WriteLine("{0}*{1}={2}", a, b, a*b);
      Console.WriteLine("{0}-{1}={2}", a, b, a-b);
      Console.WriteLine("{0}/{1}={2}", a, b, a-b);
   }
}
```

Рассмотрим как происходит работа со статическим классом:

```
class Program
{
   static void Main()
   {
      //ошибка: создать экземпляр статического класса нельзя
      //Demo S=new Demo();
      Demo.Print();
   }
}
```

Результат работы программы:

```
10+2=12
10*2=20
10/2=5
```

В версию 2.0 введена возможность описывать статический класс, то есть класс с модификатором static. Экземпляры такого класса создавать запрещено, и, кроме этого, от него запрещено наследовать. Все члены такого класса должны явным образом объявляться с модификатором static (константы и вложенные типы классифицируются как статические элементы автоматически). Конструктор для статического класса в явном виде задавать запрещается.

Замечание

Мы вернемся к проблеме создания конструкторов в разделе «Иерархия классов».

Деструкторы

В C# существует специальный вид метода, называемый деструктором, который вызывается сборщиком мусора непосредственно перед удалением объекта из памяти.

Замечание

Напоминаем, что сборщик мусора удаляет объекты, на которые нет ссылок. Он работает в соответствии со своей внутренней стратегией в неизвестные для программиста моменты времени.

Обычно в деструкторе описываются действия, гарантирующие корректность последующего удаления объекта. Например, обычно проверяется, все ли ресурсы, используемые объектом, освобождены (файлы закрыты, удаленное соединение разорвано и т.п.). Если же ваш класс не используется для обеспечения доступа к какому-либо системному ресурсу, то использование деструкторов настоятельно не рекомендуется.

Синтаксис деструктора:

```
[атрибуты] [extern] ~имя_класса()
{
   тело_деструктора
}
```

Деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется тильдой (~), символизирующей обратные по отношению к конструктору действия. Тело деструктора представляет собой блок, или просто точку с запятой. Добавим в класс Circle следующий деструктор:

```
~Circle()
{
   Console.WriteLine("Удалена {0} с центром  в точке ({1},{2}),
                      радиусом {3}", name, x, y, radius);
}
```

Теперь посмотрим, как выполнится следующая программа:

```
using System;
namespace MyProgram
{
   class Program
   {
      static void Main()
      {
         Circle oneCircle = new Circle(1);
         oneCircle.Show();
         Circle twoCircle = new Circle(1, 1, 100);
         twoCircle.Show();
         Circle threeCircle = new Circle(5, 2, 10);
         threeCircle.Show();
      }
   }
}
```

Результат работы программы:

```
Окружность с центром  в точке (0, 0) радиусом 1
```

```
Окружность с центром  в точке (1, 1) радиусом 100
Окружность с центром  в точке (5, 2) радиусом 10
Удалена Окружность с центром  в точке (5, 2) радиусом 10
Удалена Окружность с центром  в точке (1, 1) радиусом 100
Удалена Окружность с центром  в точке (0, 0) радиусом 1
```

Задание

Обратите внимание на то, что деструкторы были вызваны автоматически. Объясните, почему они были вызваны именно в такой последовательности.

Вернемся к проблеме потерянных ссылок на объекты:

```
using System;
namespace MyProgram
{
   class Program
   {
      static void Main()
      {
         //создаем три объекта
         Circle oneCircle = new Circle(1);
         Circle twoCircle = new Circle(1, 1, 100);
         Circle threeCircle = new Circle(5, 2, 10);
         //выводим информацию об объектах
         oneCircle.Show();
         twoCircle.Show();
         threeCircle.Show();
         Console.WriteLine();
         // теперь threeCircle и oneCircle ссылаются на один
         // и тот же объект; к объекту,
         // представляющему окружность с центром (0, 0) и
         // радиусом 1 не ведет ни одной ссылки
         threeCircle = oneCircle;
         //изменяем значения по ссылке threeCircle
         threeCircle.Set(-1, -1, 20);
         //вновь выводим информацию об объектах
         oneCircle.Show();
         twoCircle.Show();
         threeCircle.Show();
         Console.WriteLine();
      }
   }
}
```

Результат работы программы:

```
Окружность с центром  в точке (0, 0) радиусом 1
Окружность с центром  в точке (1, 1) радиусом 100
Окружность с центром  в точке (5, 2) радиусом 10
Окружность с центром  в точке (-1, -1) радиусом 20
Окружность с центром  в точке (1, 1) радиусом 100
Окружность с центром  в точке (-1, -1) радиусом 20
Удалена Окружность с центром  в точке (5, 2) радиусом 10
```

```
Удалена Окружность с центром  в точке (1, 1) радиусом 100
Удалена Окружность с центром  в точке (-1, -1) радиусом 20
```

Данный пример показывает, что ссылки oneCircle и threeCircle действительно ссылаются на один и тот же объект. Ссылка на окружность с центром (0, 0) и радиусом 1 потеряна. Однако сборщик мусора удалит все объекты, в том числе, и те, ссылки на которые были потеряны.

Свойства

Как мы уже отмечали ранее, члены-данные по умолчанию считаются закрытыми, т.е., для них установлен спецификатор private. Для полей этот вид доступа является предпочтительным, поскольку данные определяют внутреннее строение класса, которое должно быть скрыто от пользователя.

Для того чтобы определить для пользователя способ доступа к полям класса, можно использовать такой функциональный член класса как свойство. Свойство позволяет сделать поле доступным только для чтения, или только для записи (при этом можно проверить допустимость присваиваемых полю значений).

Синтаксис свойства:

```
[атрибуты] [спецификаторы] тип имя_свойства
{
   [get код_доступа_для_чтения]
   [set код_доступа_для записи]
}
```

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые, т.е., со спецификатором public.

Код доступа представляет собой блоки операторов, которые выполняются при получении (get), или при установке (set) свойства. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, то свойство доступно только для чтения. Если отсутствует часть get, то свойство доступно только для записи. Блок get должен содержать оператор return, возвращающий выражение, для типа которого должно существовать неявное преобразование к типу свойства. Обратиться к свойству для получения значения можно, например, следущим образом:

```
x = имя_класса.имя_свойства;
```

Обратиться к свойству для установки значения можно, например, следущим образом:

```
имя_класса.имя_свойства = значение;
```

При этом *значение* будет записано в параметр value, который будет использоваться в свойстве для установки соответствующего значения поля. При необходимости, можно осуществлять проверку допустимости устанавливаемых значений.

Внесем следующие изменения в класс Circle:

1) сделаем поля x, y, radius, name закрытыми;
2) создадим свойства для работы с закрытыми полями x, y, radius, позволяющие получать и устанавливать значения соответствующих полей; для поля radius в блоке set будем проводить проверку допустимости устанавливаемого значения;
3) создадим свойство для работы с полем name, доступным только для чтения;
4) в конструкторе 1 инициализацию поля radius будем проводить через соответствующее свойство для проверки допустимости устанавливаемого значения;

5) создадим свойство, доступное только для чтения, позволяющее определять площадь круга.

```
using System;
namespace MyProgram
{
   class Circle
   {
      //теперь поля закрыты и доступ из другого класса
      //к ним невозможен
      private int x;
      private int y;
      private int radius;
      static readonly string name="Окружность";
      public Circle( int r)
      {
         //устанавливаем поле radius через одноименное свойство,
         // для проверки допустимости
         // устанавливаемого значения
         Radius=r;
      }
      public Circle (int x, int y, int r)
      :this (r)
      {
         this.x = x;
         this.y = y;
      }
      public void Show()
      {
         Console.WriteLine("{0} с центром  в точке ({1},{2}),
                           радиусом {3}", name, x, y, radius);
      }
      public int X //свойство для обращения к закрытому полю x
      {
         get
         {
            return x;
         }
         set
         {
            x = value;
         }
      }
      public int Y //свойство для обращения к полю y
      {
         get
         {
            return y;
         }
         set
         {
            y = value;
```

```
            }
        }
        public int Radius  //свойство для обращения к полю radius
        {
            get
            {
                return radius;
            }
            set
            {
                if (value>0)
                {
                    radius = value;
                }
                else
                {
                    radius=0;
                    Console.WriteLine("Недопустимое значение для радиуса
                                       окружности {0}", value);
                }
            }
        }
        //свойство для работы с закрытым readonly полем
        public string Name
        {
            get
            {
                return name;
            }
        }
        public double Area  // свойство доступное только для чтения
        {
            get
            {
                return Math.PI*radius*radius;
            }
        }
    }
}
```

Замечания

1. *Свойство Area не пытается получить, или установить значение какого-либо закрытого поля класса. Оно решает самостоятельную задачу – вычисляет площадь круга, т.е., выполняет функциональность, характерную, скорее, для метода.*
2. *Данный пример при обнаружении ошибочных данных выводит сообщение напрямую на консоль. В реальных приложениях это недопустимо. Для сообщения о возникновении ошибки рекомендуется применять механизм обработки исключений, который будет рассмотрен позднее.*

Рассмотрим использование разработанных свойств на примере:

```
using System;
namespace MyProgram
{
   class Program
   {
      static void Main()
      {
         Circle oneCircle = new Circle(-1);
         oneCircle.Show();
         Console.WriteLine();
         oneCircle.X=-1; // параметр value принимает значение -1
         oneCircle.Y=1;  // параметр value принимает значение 1
         oneCircle.Radius=2; // параметр value принимает значение 2
         oneCircle.Show();
         Console.WriteLine("Заданная {0} имеет площадь {1:f}",
                           oneCircle.Name, oneCircle.Area);
      }
   }
}
```

Результат работы программы:

```
Недопустимое значение для радиуса окружности -1
Окружность с центром  в точке (0, 0) радиусом 0
Окружность с центром  в точке (-1, 1) радиусом 2
Заданная Окружность имеет площадь 12.56
```

Задание

Попытайтесь обратиться к свойствам Name и Area с целью установки значения. Посмотрите, какое сообщение выдаст при этом компилятор.

В .NET Framework 2.0 появилась возможность задавать отдельные права доступа для каждого из блоков set и get. Например, если для каких-либо действий необходимо задавать поле Area принудительно (изменяя при этом радиус), но не нужно, чтобы это мог делать пользователь, можно использовать следующую конструкцию:

```
public double Area
{
   get
   {
      return Math.PI * radius * radius;
   }
   private set
   {
      r = Math.sqrt(value / (Math.PI));
   }
}
```

Начиная с версии .NET Framework 3.0 можно использовать упрощенный синтаксис для свойств, не реализующих никакой логики. Например, вместо

```
private int y;
public int Y //свойство для обращения к полю y
{
   get
   {
      return y;
   }
   set
   {
      y = value;
   }
}
```

Можно писать просто:

```
public int Y { get; set; }
```

Среда исполнения сама, в процессе компиляции, создаст необходимое поле и пропишет код для его изменения.

Индексаторы

Индексатор представляет собой разновидность свойства и позволяет организовать доступ к скрытым полям класса по индексу также, как мы обращаемся к элементу массива. Синтаксис индексатора аналогичен синтаксису свойства:

```
// последние [ ] являются обязательными элементами синтаксиса
[атрибуты] [спецификаторы] тип this [список параметров]
{
   [get код_доступа]
   [set код_доступа]
}
```

Спецификаторы аналогичны спецификаторам свойств и методов. Чаще всего индексаторы объявляются со спецификатором public.

Список параметров содержит одно или несколько описаний индексов, по которым выполняется доступ к элементу. Чаще всего используется один индекс целого типа, но можно объявлять и другие типы индексов, например, строковые.

Код доступа представляет собой блоки операторов, которые выполняются при получении (get) или установке (set) значения некоторого элемента класса. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, индексатор доступен только для чтения, а если часть get – то только для записи.

В качестве примера рассмотрим индексатор, который позволяет осуществить доступ к координатам центра окружности: индекс 1 соответствует координате x, индекс 2 – координате y:

```
public int this[int i]
{
   get
   {
      if (i==1)
      {
         return x;
      }
      else
```

```
            {
               if (i==2)
               {
                  return y;
               }
               else
               {
                  Console.WriteLine("Недопустимый индекс");
                  return 0;
               }
            }
         }
         set
         {
            if (i == 1)
            {
               x = value;
            }
            else
            {
               if (i == 2)
               {
                  y = value;
               }
               else
               {
                  Console.WriteLine("Недопустимый индекс");
               }
            }
         }
      }
```

Использовать индексатор можно следующим образом:

```
Circle oneCircle = new Circle(1);
oneCircle[1] = 4;
oneCircle[2] = 5;
Console.WriteLine("Центр окружности ({0}, {1})",
                     oneCircle[1], oneCircle[2]);
```

Результат работы фрагмента программы:

```
Центр окружности (4, 5)
```

Задание

Объясните, что произойдет при выполнении команды oneCircle[3]=100;

Пример, который мы рассмотрели, носит учебный характер. На практике индексаторы очень удобно применять для создания специализированных массивов, на которые накладываются какие-либо ограничения. Рассмотрим в качестве примера класс-массив, значения элементов которого находятся в диапазоне [0, 100].

```
using System;
namespace MyProgram
{
```

```
class DemoArray
{
   int[] MyArray;  //закрытый массив
   //конструктор, использующий переменное количество аргументов
   public DemoArray(params int []array)
   {
      MyArray = new int[array.Length];
      Array.Copy(array,MyArray, array.Length);
   }
   public void Show()
   {
      foreach (int item in MyArray)
      {
         Console.Write("{0} ", item);
      }
      Console.WriteLine();
   }
   public int this[int i]  //индексатор
   {
      get
      {
         if (i <0 || i >= MyArray.Length)
         {
            Console.WriteLine("Индекс {0} выходит за границы
                               массива", i);
            return 0;
         }
         else
         {
            return MyArray[i];
         }
      }
      set
      {
         if (i < 0 || i >= MyArray.Length)
         {
            Console.WriteLine("Индекс {0} выходит за границы
                               массива", i);
         }
         else
         {
            if (value >= 0 && value <= 100)
            {
               MyArray[i] = value;
            }
            else
            {
               Console.WriteLine("Присваивается недопустимое
                                  значение {0}", value);
            }
         }
      }
```

```
        }
    }
}
```

Использовать индексатор можно следующим образом:

```
using System;
namespace MyProgram
{
    class Program
    {
        static void Main()
        {
            DemoArray ob = new DemoArray(2, 6, 7, 8, 1, 3, 0, 8, 4);
            ob.Show();
            ob[2] = 100;
            ob[3] = 200;
            ob[100] = 10;
            ob.Show();
        }
    }
}
```

Результат работы программы:

```
2 6 7 8 1 3 0 8 4
Присваивается недопустимое значение 200
Индекс 100 выходит за границы массива
2 6 100 8 1 3 0 8 4
```

Язык C# допускает использование многомерных индексаторов, которые могут использоваться, например, для многомерных массивов. Синтаксис описания многомерного индексатора:

```
public int this[int i, int j, …]  // многомерный индексатор
{
    get
    {
        …
    }
    set
    {
        …
    }
}
```

Аналогичным образом в качестве параметров индексатора можно использовать не только целые числа, но и строки, или любые другие программные объекты.

Задание

В предыдущем примере преобразуйте класс DemoArray для работы с двумерным массивом и определите для него двумерный индексатор. Проиллюстрируйте работу с данным индексатором на примере.

Операции класса

C# позволяет переопределить большинство операций так, чтобы при использовании их объектами конкретного класса выполнялись действия, отличные от стандартных. Это дает возможность применять объекты собственных типов данных в составе выражений, например:

```
newObject x, y, z;
…
// используется операция сложения, переопределенная для
// класса newObject
z = x+y;
```

Определение собственных операций класса называют перегрузкой операций. Перегрузка операций обычно применяется для классов, для которых семантика операций делает программу более понятной. Если назначение операции не понятно интуитивно, перегружать такую операцию не рекомендуется.

Операции класса описываются с помощью методов специального вида, синтаксис которых выглядит следующим образом:

```
[ атрибуты] спецификаторы объявитель_операции
{
   тело
}
```

В качестве спецификаторов одновременно используются ключевые слова public и static. Кроме того, операцию можно объявить как внешнюю (extern). Объявление операции может выглядеть по-разному, в зависимости от того, какую операцию мы перегружаем: унарную, или бинарную, но в любом случае:

1) операция должна быть описана как открытый статический метод класса (public static);
2) параметры в операцию должны передаваться по значению (то есть недопустимо использовать параметры ref, out, или params);
3) допустимо определять несколько вариантов перегруженных операций класса, но их сигнатуры (заголовки) должны различаться;
4) типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

Унарные операции

В классе можно переопределять следующие унарные операции: + – ! ~ ++ —, а также константы true и false. При этом, если была перегружена константа true, то должна быть перегружена и константа false, и наоборот.

Синтаксис объявления унарной операции:

```
тип operator унарная_операция (параметр)
```

Примеры заголовков унарных операций:

```
public static int operator + (DemoArray m)
public static DemoArray operator --(DemoArray m)
public static bool operator true (DemoArray m)
```

Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. При этом операция должна возвращать:

1) для операций +, –, !, ~ величину любого типа;
2) для операций ++, — величину типа класса, для которого она определяется;
3) для операций true и false величину типа bool.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

В качестве примера рассмотрим класс DemoArray, реализующий одномерный массив, в котором содержатся следующие функциональные элементы:

1) конструктор, позволяющий создать объект-массив на основе последовательности чисел переменной длины;
2) конструктор, позволяющий инициализировать объект-массив на основе уже существующего объекта-массива;
3) метод, выводящий объект-массив на экран;
4) свойство, возвращающее размерность массива;
5) индексатор, позволяющий просматривать и устанавливать значение по индексу в закрытом поле-массиве;
6) перегрузка операции унарный минус (все элементы массива меняют свое значение на противоположное);
7) перегрузка операции инкремента (все элементы массива увеличивают свое значение на 1);
8) перегрузка констант true и false (при обращении к объекту будет возвращаться значение true, если все элементы массива положительные, в противном случае, будет возвращаться значение false).

```
using System;
namespace MyProgram
{
   class DemoArray
   {
      int[] MyArray;
      public DemoArray(params int []array)  //конструктор 1
      {
         MyArray = new int[array.Length];
          Array.Copy(array,MyArray, array.Length);
      }
      public DemoArray(DemoArray array)     //конструктор 2
      {
         MyArray = new int[array.Length];
          Array.Copy(array,MyArray, array.Length);
      }
      public void Show()
      {
         foreach (int item in MyArray)
         {
            Console.Write("{0} ", item);
         }
```

```
            Console.WriteLine();
        }
        //свойство, возвращающее размерность массива
        public int Length
        {
            get { return MyArray.Length; }
        }
        public int this[int i]  //индексатор
        {
            get
            {
                if (i < 0 || i >= MyArray.Length)
                {
                    Console.WriteLine("Индекс {0} выходит за границы
                                       массива", i);
                    return 0;
                }
                else
                {
                    return MyArray[i];
                }
            }
            set
            {
                if (i < 0 || i >= MyArray.Length)
                {
                    Console.WriteLine("Индекс {0} выходит за границы
                                       массива", i);
                }
                else
                {

                    MyArray[i] = value;
                }
            }
        }
        //перегрузка операции унарный минус
        public static DemoArray operator - (DemoArray x)
        {
            DemoArray temp = new DemoArray(x);
            for (int i = 0; i < x.Length; i++)
            {
                temp[i] = -x[i];
            }
            return temp;
        }
        //перегрузка операции инкремента
        public static DemoArray operator ++ (DemoArray x)
        {
            DemoArray temp = new DemoArray(x);
```

```
            for (int i = 0; i < x.Length; i++)
               temp[i] = x[i]+1;
            return temp;
         }
         //перегрузка константы true
         public static bool operator true(DemoArray a)
         {
            foreach (int item in a.MyArray)
            {
               if (i<0)
               {
                  return false;
               }
            }
            return true;
         }
         //перегрузка константы false
         public static bool operator false(DemoArray a)
         {
            foreach (int item in a.MyArray)
            {
               if (i>0)
               {
                  return true;
               }
            }
            return false;
         }
      }
   }
```

Продемонстрируем работу с данным классом:

```
using System;
namespace MyProgram
{
   class Program
   {
      static void Main()
      {
         DemoArray oneArray = new DemoArray(1, -4, 3, -5, 0);
         Console.Write("Первый массив: ");
         oneArray.Show();
         Console.WriteLine();

         Console.WriteLine("Унарный минус");
         DemoArray twoArray=-oneArray;
         Console.Write("Первый массив: ");
         oneArray.Show();
         Console.Write("Второй массив ");
         twoArray.Show();
         Console.WriteLine();

         Console.WriteLine("Операция префиксного инкремента");
```

```
            ++oneArray;              //1
            Console.Write("Первый массив: ");
            oneArray.Show();
            Console.WriteLine();

            Console.WriteLine("Операция постфиксного инкремента");
            DemoArray threeArray = oneArray++; //2
            Console.Write("Первый массив: ");
            oneArray.Show();
            Console.Write("Третий массив: ");
            threeArray.Show();
            Console.WriteLine();

            //проверка на положительность элементов массива
            if (oneArray)
               Console.WriteLine("В первом массиве все
                                  элементы положительные");
            else
               Console.WriteLine("В первом массиве есть не
                                  положительные элементы");
         }
      }
   }
```

Результат работы программы:

```
Первый массив: 1 -4 3 -5 0

Унарный минус
Первый массив: 1 -4 3 -5 0
Второй массив: -1 4 -3 5 0

Операция префиксного инкремента
Первый массив: 2 -3 4 -4 1

Операция постфиксного инкремента
Первый массив: 3 -2 5 -3 2
Третий массив: 2 -3 4 -4 1

В первом массиве есть не положительные элементы
```

Замечание

Обратите внимание на то, что перегруженную операцию инкремента можно использовать как в качестве самостоятельной операции (строка 1), так и в составе выражения (строка 2).

Задание

Добавьте в класс DemoArray переопределение унарного плюса (все элементы массива преобразуются в положительные) и унарного декремента (все элементы массива уменьшаются на единицу). Продемонстрируйте работу перегруженных операций на примере.

Бинарные операции

При разработке класса можно перегрузить следующие бинарные операции: + – * / % & | ^ << >> == != < > <= >=. Обратите внимание: операций присваивания в этом списке нет.

Синтаксис объявителя бинарной операции:

```
тип operator бинарная_операция (параметр1, параметр 2)
```

Примеры заголовков бинарных операций:

```
public static DemoArray operator + (DemoArray a, DemoArray b)
public static bool operator == (DemoArray a, DemoArray b)
```

При переопределении бинарных операций нужно учитывать ряд правил.

1. Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется.
2. Операция может возвращать величину любого типа.
3. Операции отношений определяются только парами и обычно возвращают логическое значение. Чаще всего переопределяются операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение значения некоторых полей объектов, а не ссылок на объект. Для того, чтобы переопределить операции отношений, требуется знание стандартных интерфейсов, которые будут рассматриваться чуть позже.

В качестве примера вернемся к классу DemoArray, реализующему одномерный массив, и добавим в него две версии переопределенной операции +:

- Вариант 1: добавляет к каждому элементу массива заданное число;
- Вариант 2: поэлементно складывает два массива

```
//вариант 1
public static DemoArray operator +(DemoArray x, int a)
{
   DemoArray temp = new DemoArray(x);
   for (int i = 0; i < x.Length; i++)
   {
      temp[i] = x[i] + a;
   }
   return temp;
}
//вариант 2
public static DemoArray operator + (DemoArray x, DemoArray y)
{
   if (x.Length == y.Length)
   {
      DemoArray temp = new DemoArray(x);
      for (int i = 0; i < x.Length; i++)
      {
         temp[i] = x[i] + y[i];
      }
      return temp;
   }
   else
   {
      Console.WriteLine("Несоответствие размерностей массивов");
      return null;
   }
}
```

Рассмотрим на фрагменте программы пример использования перегруженной операции сложения:

```
DemoArray oneArray = new DemoArray(1, -4, 3, -5, 0);
DemoArray twoArray = new DemoArray(3, 6, 10, 0, -2);
DemoArray threeArray = oneArray + twoArray;
DemoArray fourArray = oneArray + 5;
Console.Write("Первый массив: ");
oneArray.Show();
Console.Write("Второй массив: ");
twoArray.Show();
Console.Write("Третий массив: ");
threeArray.Show();
Console.Write("Четвертый массив: ");
fourArray.Show();
```

Результат работы фрагмента программы:

```
Первый массив:  1  -4  3  -5  0
Второй массив:  3  6  10  0  -2
Третий массив:  4  2  13  -5  -2
Четвертый массив:  6  1  8  0  5
```

Задание

Добавьте в класс DemoArray переопределение бинарного минуса (из всех элементов массива вычитается заданное число) и операции & (поэлементно сравнивает два массива: если соответствующие элементы попарно совпадают, то операция возвращает значение true, иначе false).

Операции преобразования типов

Операции преобразования типов обеспечивают возможность явного и неявного преобразования между пользовательскими типами данных. Синтаксис описания операции преобразования типов выглядит следующим образом:

```
explicit operator целевой_тип (параметр)    //явное преобразование
implicit operator целевой_тип (параметр)    //неявное преобразование
```

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого выполняется преобразование.

Неявное преобразование выполняется автоматически в следующих ситуациях:

1) при присваивании объекта переменной целевого типа;
2) при использовании объекта в выражении, содержащем переменные целевого типа;
3) при передаче объекта в метод параметра целевого типа;
4) при явном приведении типа.

Явное преобразование выполняется при использовании операции приведения типа.

При определении операции преобразования типа следует учитывать следующие особенности:

1) тип возвращаемого значения (целевой_тип) включается в сигнатуру объявителя операции;
2) ключевые слова explicit и implicit не включаются в сигнатуру объявителя операции.

Следовательно, для одного и того класса нельзя определить одновременно и явную, и неявную версию. Однако, т.к. неявное преобразование автоматически выполнятся при явном использовании операции приведения типа, то, если необходимо чтобы для проектируемого класса выполнялись обе операции преобразования, достаточно разработать только неявную версию операции.

В качестве примера вернемся к классу DemoArray, реализующему одномерный массив, и добавим в него неявную версию переопределения типа DemoArray в тип одномерный массив, и наоборот:

```
//неявное преобразование типа int [] в DemoArray
public static implicit operator DemoArray (int []a)
{
   return new DemoArray(a);
}
//неявное преобразование типа DemoArray в int []
public static implicit operator int [](DemoArray a)
{
   int []temp = new int[a.Length];
   for (int i = 0; i < a.Length; i++)
   {
      temp[i] = a[i];
   }
   return temp;
}
```

Продемонстрируем работу данных методов на примере:

```
using System;
namespace MyProgram
{
   class Program
   {
      //выводит на экран одномерный массив
      static void Print(int[]a)
      {
         for (int i = 0; i < a.Length; i++)
            Console.Write(a[i] + " ");
         Console.WriteLine();
      }
      static void Main()
      {
         DemoArray a = new DemoArray(1, -4, 3, -5, 0);
         //неявное преобразование типа DemoArray в int []
         int[] mas1 = a;

         //явное преобразование типа DemoArray в int []
         int[] mas2 = (int[]) a;

         //неявное преобразование типа int [] в DemoArray
         DemoArray b1 = mas1;

         //явное преобразование типа int [] в DemoArray
```

```
            DemoArray b2 = (DemoArray)mas2;

            //изменение значений
            mas1[0] = 0;
            mas2[0] = -1;
            b1[0] = 100;
            b2[0] = -100;

            //вывод данных на экран
            Console.Write("Массива a: ");
            a.Show();
            Console.Write("Массив mas1: ");
            Print(mas1);
            Console.Write("Массив mas2: ");
            Print(mas2);
            Console.Write("Массив b1: ");
            b1.Show();
            Console.Write("Массив b2: ");
            b2.Show();
        }
    }
}
```

Результат работы программы:

```
Массив a:  1  -4  3  -5  0
Массив mas1:  0  -4  3  -5  0
Массив mas2:  -1  -4  3  -5  0
Массив b1:  100  -4  3  -5  0
Массив b2:  -100  -4  3  -5  0
```

Замечание

Изменения в массивы вносились для того, чтобы показать, что были созданы новые объекты, а не установлено несколько ссылок на один и тот же объект.

Задание

В методе Main используется операция приведения типа DemoArray к int[] (и наоборот) в явном виде, хотя явная версия операции преобразования типа DemoArray к int[] не была определена. Объясните, почему возможно использование явного приведения типа.

Замечание

Возможности определения сложных структур данных внутри класса будут рассмотрены позже. А изучение такого функционального члена класса как событие выходит за рамки данного пособия.

Практикум №10

Задание 1

Создать класс Point, содержащий следующие члены класса:

1. Поля:
   - int x, y;
2. Конструкторы, позволяющие создать экземпляр класса:
   - с нулевыми координатами;

- с заданными координатами.

3. Методы, позволяющие:
   - вывести координаты точки на экран;
   - рассчитать расстояние от начала координат до точки;
   - переместить точку на плоскости на вектор (a, b).
4. Свойства, позволяющие:
   - получить и установить координаты точки (доступное для чтения и записи);
   - умножить координаты точки на скаляр (доступное только для записи).
5. Индексатор, позволяющий по индексу 0 обращаться к полю x, по индексу 1 – к полю y; при других значениях индекса должно выдаваться сообщение об ошибке.
6. Перегруженные операции и константы, позволяющие:
   - одновременно увеличить (уменьшить) значения полей x и y на 1 (операции ++ и --);
   - узнать, совпадают, или нет значения полей x и y (константы true и false);
   - одновременно добавить к полям x и y значение скаляра (бинарный +).

Продемонстрировать работу класса.

Задание 2

Создать класс Triangle, содержащий следующие члены класса:

1. Поля:
   - int a, b, c;
2. Конструктор, позволяющий создать экземпляр класса с заданными длинами сторон.
3. Методы, позволяющие:
   - вывести длины сторон треугольника на экран;
   - расчитать периметр треугольника;
   - расчитать площадь треугольника.
4. Свойства, позволяющие:
   - получить и установить длины сторон треугольника (доступные для чтения и записи);
   - установить, существует ли треугольник с данными длинами сторон (доступное только для чтения).
5. Индексатор, позволяющий по индексу 0 обращаться к полю a, по индексу 1 – к полю b, по индексу 2 – к полю c; при других значениях индекса должно выдаваться сообщение об ошибке.
6. Перегруженные операции и константы, позволяющие:
   - одновременно увеличить (уменьшить) значения полей a, b и c на 1 (операции ++ и --);
   - узнать, существует треугольник с заданными сторонами, или нет (константы true и false);
   - одновременно умножить поля a, b и c на скаляр (операция *).

Продемонстрировать работу класса.

Задание 3

Создать класс Rectangle, содержащий следующие члены класса:

1. Поля:
   - int a, b;
2. Конструктор, позволяющий создать экземпляр класса с заданными длинами сторон.
3. Методы, позволяющие:
   - вывести длины сторон прямоугольника на экран;
   - расчитать периметр прямоугольника;
   - расчитать площадь прямоугольника.
4. Свойства, позволяющие:
   - получить и установить длины сторон прямоугольника (доступные для чтения и записи);
   - установить, является ли данный прямоугольник квадратом (доступное только для чтения).
5. Индексатор, позволяющий по индексу 0 обращаться к полю a, по индексу 1 – к полю b; при других значениях индекса должно выдаваться сообщение об ошибке.
6. Перегруженные операции и константы, позволяющие:
   - одновременно увеличить (уменьшить) значение полей a и b на 1 (операции ++ и --);
   - узнать, является ли прямоугольник квадратом (константы true и false);
   - одновременно умножить поля a и b на скаляр (операция *).

Продемонстрировать работу класса.

Задание 4

Создать класс Money, содержащий следующие члены класса:

1. Поля:
   - int first; //номинал купюры
   - int second; //количество купюр
2. Конструктор, позволяющий создать экземпляр класса с заданными значениям полей.
3. Методы, позволяющие:
   - вывести номинал и количество купюр;
   - определить, хватит ли денежных средств на покупку товара на сумму N рублей.
   - определить, сколько штук товара стоимости n рублей можно купить на имеющиеся денежные средства.
4. Свойства, позволяющие:
   - получить и установить значение полей (доступные для чтения и записи);
   - расчитать сумму денег (доступное только для чтения).
5. Индексатор, позволяющий по индексу 0 обращаться к полю first, по индексу 1 – к полю second; при других значениях индекса должно выдаваться сообщение об ошибке.
6. Перегруженные операции:
   - ++ и -- для одновременно увеличения и уменьшения значений полей first и second;
   - бинарный + для увеличения значения поля second на скалярную величину.

Продемонстрировать работу класса.

Задание 5

Создать класс для работы с одномерным массивом целых чисел. Разработать следующие члены класса:

1. Поля:
   - int [] IntArray;
2. Конструктор, позволяющий создать массив размерности n.
3. Методы, позволяющие:
   - ввести элементы массива с клавиатуры;
   - вывести элементы массива на экран;
   - отсортировать элементы массива в порядке возрастания.
4. Свойства:
   - возвращающее размерность массива (доступное только для чтения);
   - позволяющее домножить все элементы массива на скаляр (доступное только для записи).
5. Индексатор, позволяющий по индексу обращаться к соответствующему элементу массива.
6. Перегруженные операции:
   - ++ и -- для одновременного увеличения и уменьшения значений всех элементов массива на 1;
   - бинарное *, для умножения всех элементы массива на скаляр;
   - для преобразования экземпляра класса в одномерный массив (и наоборот).

Продемонстрировать работу класса.

Задине 6

Создать класс для работы с двумерным массивом целых чисел. Разработать следующие члены класса:

1. Поля:
   - int [,] intArray;
2. Конструктор, позволяющий создать массив размерности n×m.
3. Методы, позволяющие:
   - ввести элементы массива с клавиатуры;
   - вывести элементы массива на экран;
   - вычислить сумму элеметов столбца с номером i.
4. Свойства, позволяющие:
   - вычислить количество нулевых элементов в массиве (доступное только для чтения);
   - установить значение всех элементов главной диагонали массива равное скаляру (доступное только для записи).
5. Двумерный индексатор, позволяющий обращаться к соответствующему элементу массива.
6. Перегруженные операции:

- ++ и -- для одновременного увеличения и уменьшения значения всех элементов массива на 1;
- бинарный + для сложения двух массивов одинакового размера;
- для преобразования экземпляра класса в двумерный массив (и наоборот).

Продемонстрировать работу класса.

Задание 7

Создать класс для работы со ступенчатым массивом вещественных чисел. Разработать следующие функциональные члены класса:

1. Поля:
   - double [][] doubelArray;
2. Конструктор, позволяющий создать ступенчатый массив.
3. Методы, позволяющие:
   - ввести элементы массива с клавиатуры;
   - вывести элементы массива на экран;
   - отсортировать элементы каждой строки массива в порядке убывания.
4. Свойства:
   - возвращающее общее количество элементов в массиве (доступное только для чтения);
   - позволяющее увеличить значение всех элементов массива на скаляр (доступное только для записи).
5. Двумерный индексатор, позволяющий обращаться к соответствующему элементу массива.
6. Перегруженные операции и константы, позволяющие:
   - увеличить, или уменьшить значение всех элементов массива на 1 (++ и --);
   - проверить, является ли каждая строка массива упорядоченной по возрастанию (true и false);
   - преобразовать экземпляр класса в ступенчатый массив (и наоборот).

Продемонстрировать работу класса.

Задание 8

Создать класс для работы со строками. Разработать следующие члены класса:

1. Поле:
   - string line;
2. Конструктор, позволяющий создать строку на основе заданного строкового литерала.
3. Методы, позволяющие:
   - подсчитать количество цифр в строке;
   - вывести на экран все символы строки, встречающиеся в ней ровно один раз;
   - вывести на экран самую длинную последовательность повторящихся символов в строке.
4. Свойство, возвращающее общее количество символов в строке (доступное только для чтения);

5. Индексатор, позволяющий по индексу обращаться к соответствующему символу строки (доступный только для чтения).
6. Перегруженные операции и константы, позволяющие:
   - проверить, является ли строка палиндромом(константы true и false);
   - узнать, являются ли строки равными посимвольно, без учета регистра (операции == и !=);
   - преобразовать экземпляр класса в тип string (и наоборот).

Продемонстрировать работу класса.

Задание 9

Создать класс для работы со строками. Разработать следующие члены класса:

1. Поле:
   - StringBuilder line;
2. Конструктор, позволяющий создать строку на основе заданного строкового литерала, и конструктор, позволяющий создавать пустую строку.
3. Методы, позволяющие:
   - подсчитать количество пробелов в строке;
   - заменить в строке все прописные символы на строчные;
   - удалить из строки все знаки препинания.
4. Свойства:
   - возвращающее общее количество элементов в строке (доступное только для чтения);
   - позволяющее установить значение поля line, в соответствии с введенным значением строки с клавиатуры, а также получить значение данного поля (доступно для чтения и записи)
5. Индексатор, позволяющий по индексу обращаться к соответствующему символу строки.
6. Перегруженные операции и константы, позволяющие:
   - преобразовать строку к верхнему и нижнему регистрам (операции унарного + и –);
   - проверить, является ли строка не пустой (константы true и false).
   - узнать, являются ли строки равными посимвольно, без учета регистра (операции == и !=);
   - преобразовать экземпляр класса в тип StringBuilder (и наоборот).

Продемонстрировать работу класса.

Задание 10

Самостоятельно изучите тип данных DateTime, на основе которого необходимо создать класс для работы с датой. Данный класс должен содержать следующие члены класса:

1) Поле:
   - DateTime data.
2) Конструкторы, позволяющие установить:
   - заданную дату
   - дату 1.01.2010

3) Методы, позволяющие:
   - вычислить дату предыдущего дня;
   - вычислить дату следующего дня;
   - определить, сколько дней осталось до конца месяца.
4) Свойства, позволяющие:
   - установить, или получить значение поле класса (доступно для чтения и записи)
   - определить, является ли год высокосным (доступно только для чтения).
5) Индексатор, позволяющий определить дату i-того по счету дня относительно установленной даты (при отрицательных значениях индекса отсчет должен вестись в обратном порядке).
6) Перегруженные операции и константы, позволяющие определить:
   - является ли установленная дата началом года (константы true и false);
   - равны ли две даты (операции == и !=).

Продемонстрировать работу класса.

Структуры

Классы, как вы уже знаете, являются ссылочными типами данных. Это означает, что к экземплярам классов можно обратиться только через ссылку. В C# реализован тип данных, аналогичный классу, но являющийся размерным типом. Таким типом является структура.

Так как структура является размерным типом, то экземпляр структуры хранит значение своих элементов, а не ссылки на них. В связи с этим фактом структура не может участвовать в наследовании, а может только реализовывать интерфейсы. Кроме того, структуре запрещено:

1) определять конструктор по умолчанию, поскольку он определен неявно и присваивает всем своим элементам значения по умолчанию (нули соответствующего типа);
2) определять деструктор, поскольку это бессмысленно.

Синтаксис структуры:

```
[атрибуты][спецификаторы] struct имя_структуры [: интерфейсы]
{
   тело_структуры
}
```

Спецификаторы структуры имеют такой же смысл, как и для класса. Однако из спецификаторов доступа допускается использовать только public, internal и, для вложенных структур, еще и private. Кроме того, структуры не могут быть абстрактными, поскольку они не могут иметь потомков.

Интерфейсы, реализуемые структурой, перечисляются через запятую.

Замечание

Работа с интерфейсами будет рассмотрена позже.

Тело структуры может содержать: константы, поля, конструкторы, методы, свойства, индексаторы, операторные методы, вложенные типы и события.

При описании структуры задавать значение по умолчанию можно только для статических полей. Остальным полям с помощью конструктора по умолчанию будут присвоены нули для полей размерных типов и null для полей ссылочных типов.

Параметр this интерпретируется как значение, поэтому его можно использовать для ссылок, но не для присваивания.

Так как структуры не могут участвовать в иерархии, то для их членов недопустимо использовать спецификаторы protected и protected internal. Методы структур не могут быть абстрактными и виртуальными, а переопределяться могут только те методы, которые унаследованы от базового класса System.Object.

Рассмотрим пример структуры SPoint, в которой определены:

1) поля x и y;
2) конструктор;
3) метод Show, выводящий данные о точке на экран;
4) метод Distance, вычисляющий расстояние от точки до начала координат.

Используя данную структуру, рассмотрим следующую задачу. Пусть в файле text.txt хранятся координаты N точек на плоскости. Необходимо считать данные о точках из файла и вывести их на экран в порядке увеличения расстояния от точек до начала координат.

```
using System;
using System.IO;
namespace MyProgram
{
   struct SPoint   //описание структуры
   {
      public int x, y;
      public SPoint (int x, int y)
      {
         this.x = x;
         this.y = y;
      }
      public void Show()
      {
         Console.WriteLine("({0}, {1})", x, y);
      }
      public double Distance() //метод
      {
         return Math.Sqrt(x * x + y * y);
      }
   }
   class Program
   {
      static public SPoint[] Input() //читаем данные из файла
      {
         using (StreamReader fileIn =
                        new StreamReader ("d:/Example/text.txt"))
         {
            int n = int.Parse( fileIn.ReadLine());
            SPoint []ar = new SPoint[n];
            for (int i = 0; i < n; i++)
            {
               string[] text = fileIn.ReadLine().Split(' ');
               ar[i] = new SPoint(int.Parse(text[0]),
                                  int.Parse(text[1]));
```

```
            }
            return ar;
         }
      }
      static void Print (SPoint []array) //выводим данные на экран
      {
         foreach (SPoint item in array)
         {
            item.Show();
         }
      }
      static void Sort(SPoint []array) //сортируем данные массива
      {
         SPoint temp;
         for (int i = 0; i < array.Length - 1; i++)
         {
            bool isSorted = true;
            for (int j = array.Length - 1; j > i; j--)
            {
               if (array[j].Distance()< array[j - 1].Distance())
               {
                  isSorted = false;
                  temp = array[j];
                  array[j] = array[j - 1];
                  array[j - 1] = temp;
               }
            }
            if (isSorted)
               return;
         }
      }
      static void Main()
      {
         SPoint []array = Input();
         Sort(array);
         Console.WriteLine("Упорядоченные данные: ");
         Print(array);
      }
   }
}
```

text.txt

```
6
10 0
1 2
0 -1
2 4
0 5
-3 -1
```

Результат работы программы:

```
Упорядоченные данные:
(0, -1)
(1, 2)
(-3, -1)
(2, 4)
(0, 5)
(10, 0)
```

Задание

Измените программу так, чтобы на экран выводилась наиболее удаленная от начала координат точка. Алгоритм сортировки при этом не применять.

В предыдущем примере, экземпляр структуры, как и экземпляр класса, мы создавали с помощью оператора new, но это не обязательно. Если оператор new не используется, то структура все равно создается, но не инициализируется. По этой причине следующая последовательность команд будет ошибочна:

```
SPoint one;
Console.WriteLine(one.ToString());
```

Если при объявлении структуры не был вызван конструктор, то поля нужно инициализировать вручную:

```
SPoint one;
one.x = 100;
one.y = 100;
Console.WriteLine(one.ToString());
```

Так как структуры являются размерными типами, то присваивание одной структуры другой создает копию экземпляра структуры. Это важное отличие структуры от класса. Следующий фрагмент программы проиллюстрирует тот факт, что структуры one и two не зависят друг от друга:

```
SPoint one = new SPoint(); //вызов конструктора по умолчанию
SPoint two = new SPoint(1,1); //вызов собственного конструктора
one.Show();
two.Show();
one = two;
one.x = 100;
one.Show();
two.Show();
```

Результат работы фрагмента программы:

```
(0, 0)
(1, 1)
(100, 1)
(1, 1)
```

Как при решении практических задач выбрать, что использовать: класс, или структуру? Если создаваемый тип данных содержит небольшое количество полей, то расходы на выделение динамической памяти сделают использование классов неэффективным. Во всех остальных случаях лучше использовать классы, так как передача структуры в методы по значению потребует и дополнительного времени, и дополнительной памяти для создания копии.

Практикум №11

Замечания

1. *Во всех задачах данного раздела подразумевается, что исходная информация хранится в текстовом файле input.txt, каждая строка которого содержит полную информацию о некотором объекте; результирующая информация должна быть записана в файл output.txt.*
2. *Для хранения данных внутри программы организовать массив структур.*

Задание 1

На основе данных входного файла составить список студентов группы, включив следующие данные: ФИО, год рождения, домашний адрес и номер законченной школы. Вывести в новый файл информацию о студентах, окончивших заданную школу, отсортировав их по году рождения.

Задание 2

На основе данных входного файла составить список студентов группы, включив следующие данные: ФИО, номер группы и результаты сдачи трех экзаменов. Вывести в новый файл информацию о студентах, успешно сдавших сессию, отсортировав их по номеру группы.

Задание 3

На основе данных входного файла составить багажную ведомость камеры хранения, включив следующие данные: ФИО пассажира, количество вещей и их общий вес. Вывести в новый файл информацию о тех пассажирах, средний вес багажа которых превышает заданный, отсортировав их по количеству вещей, сданных в камеру хранения.

Задание 4

На основе данных входного файла составить автомобильную ведомость, включив следующие данные: марка автомобиля, номер автомобиля, фамилия его владельца, год приобретения и пробег. Вывести в новый файл информацию об автомобилях, выпущенных ранее определенного года, отсортировав их по пробегу.

Задание 5

На основе данных входного файла составить список сотрудников учреждения, включив следующие данные: ФИО, год принятия на работу, должность, зарплата и рабочий стаж. Вывести в новый файл информацию о сотрудниках, имеющих зарплату ниже определенного уровня, отсортировав их по рабочему стажу.

Задание 6

На основе данных входного файла составить инвентарную ведомость склада, включив следующие данные: вид продукции, стоимость, сорт и количество. Вывести в новый файл информацию о той продукции, количество которой менее заданной величины. Записи должны быть отсортированы по количеству продукции на складе.

Задание 7

На основе данных входного файла составить инвентарную ведомость игрушек, включив следующие данные: название игрушки, ее стоимость (в руб.), а также возрастные границы детей, для которых предназначена игрушка. Вывести в новый файл информацию о тех игрушках, которые предназначены для детей от N до M лет, отсортировав записи по стоимости.

Задание 8

На основе данных входного файла составить список вкладчиков банка, включив следующие данные: ФИО, № счета, сумма и год открытия счета. Вывести в новый файл информацию о тех вкладчиках, которые открыли вклад в текущем году, отсортировав записи по сумме вклада.

Задание 9

На основе данных входного файла составить список студентов, включающий фамилию, факультет, курс, группу и 5 оценок. Вывести в новый файл информацию о тех студентах, которые имеют хотя бы одну двойку, отсортировав записи по курсу.

Задание 10

На основе данных входного файла составить список студентов, включающий ФИО, курс, группу и результат забега. Вывести в новый файл информацию о студентах, показавших три лучших результата в забеге. Если окажется, что некоторые студенты получили такие же высокие результаты, то добавить их к списку победителей.