Тут ты найдешь инфу по ***get***, ***set***, ***properties, indexator***

Как вам должно быть уже известно, индексирование массива осуществляется с помощью оператора []. Для создаваемых классов можно определить оператор [], но с этой целью вместо операторного метода создается индексатор, который позволяет индексировать объект, подобно

массиву. Индексаторы применяются, главным образом, в качестве средства, поддерживающего создание специализированных массивов, на которые накладывается одно или несколько ограничений. Тем не менее индексаторы могут служить практически любым целям, для которых

выгодным оказывается такой же синтаксис, как и у массивов. Индексаторы могут быть одно- или многомерными.

# Создание одномерных индексаторов

Ниже приведена общая форма одномерного индексатора:

**тип\_элемента this [int индекс] {**

**// Аксессор для получения данных.**

**get {**

**// Возврат значения, которое определяет индекс.**

**}**

**// Аксессор для установки данных.**

**set {**

**// Установка значения, которое определяет индекс.**

**}**

**}**

где *тип\_элемента* обозначает конкретный тип элемента индексатора. Следовательно,

у каждого элемента, доступного с помощью индексатора, должен быть определенный

*тип\_элемента.* Этот тип соответствует типу элемента массива. Параметр *индекс* получает

конкретный индекс элемента, к которому осуществляется доступ. Формально

этот параметр совсем не обязательно должен иметь тип int, но поскольку индексаторы,

как правило, применяются для индексирования массивов, то чаще всего используется

целочисленный тип данного параметра.

В теле индексатора определены два ***аксессора*** (т.е. средства доступа к данным): ***get***

и ***set***. Аксессор подобен методу, за исключением того, что в нем не объявляется тип

возвращаемого значения или параметры. Аксессоры вызываются автоматически при

использовании индексатора, и оба получают *индекс* в качестве параметра. Так, если

индексатор указывается в левой части оператора присваивания, то вызывается аксессор

set и устанавливается элемент, на который указывает параметр *индекс.* В противном

случае вызывается аксессор get и возвращается значение, соответствующее параметру

*индекс.* Кроме того, аксессор set получает неявный параметр value, содержащий

значение, присваиваемое по указанному индексу.

В качестве

примера рассмотрим программу, в которой создается класс FailSoftArray, реализующий

массив для выявления ошибок нарушения границ массива, а следовательно,

для предотвращения исключительных ситуаций, возникающих во время выполнения

в связи с индексированием массива за его границами. Для этого массив инкапсулируется

в качестве закрытого члена класса, а доступ к нему осуществляется только с помощью

индексатора. При таком подходе исключается любая попытка получить доступ

к массиву за его границами, причем эта попытка пресекается без катастрофических

последствий для программы. А поскольку в классе FailSoftArray используется индексатор,

то к массиву можно обращаться с помощью обычной формы записи.

(glava10\_1)

class FailSoftArray

{

int[] a; //link to base array

public int Length; //length of array opened

public bool ErrFlag; //result of last operation

//make array set size

public FailSoftArray(int size)

{

a = new int[size];

Length = size;

}

//indexer for this class

public int this[int index]

{

//accessor get

get

{

if (ok(index))

{

ErrFlag = false;

return a[index];

}

else

{

ErrFlag = true;

return 0;

}

}

//accessor set

set

{

if (ok(index))

{

a[index] = value;

ErrFlag = false;

}

else ErrFlag = true;

}

}

//return true, if index is in borders

private bool ok(int index)

{

if (index >= 0 & index < Length) return true;

return false;

}

}

class FSDemo

{

static void Main()

{

FailSoftArray fs = new FailSoftArray(5);

int x;

//detec hiden erros

Console.WriteLine("Hiden error");

for (int i = 0; i < (fs.Length \* 2); i++)

fs[i] = i \* 10;

for (int i = 0; i < (fs.Length \* 2); i++)

{

x = fs[i];

if (x != -1) Console.Write(x + " ");

}

Console.WriteLine();

//Show errors

Console.WriteLine("\nError with notification");

for (int i = 0; i < (fs.Length \* 2); i++)

{

fs[i] = i \* 10;

if (fs.ErrFlag)

Console.WriteLine("fs [" + i + "] out of borders");

}

for (int i = 0; i < (fs.Length \* 2); i++)

{

x = fs[i];

fs[i] = i \* 10;

if (!fs.ErrFlag) Console.Write(x + " ");

else

Console.WriteLine("fs [" + i + "] out of borders");

}

}

}

Индексатор препятствует нарушению границ массива. Внимательно проанализируем

каждую часть кода индексатора. Он начинается со следующей строки.

public int this[int index]

В этой строке кода объявляется индексатор, оперирующий элементами типа int.

Ему передается индекс в качестве параметра index. Кроме того, индексатор объявляется

открытым (public), что дает возможность использовать этот индексатор в коде за

пределами его класса.

Рассмотрим следующий код аксессора get.

get

{

if (ok(index))

{

ErrFlag = false;

return a[index];

}

else

{

ErrFlag = true;

return 0;

}

}

Аксессор get предотвращает ошибки нарушения границ массива, проверяя в первую

очередь, находится ли индекс в установленных границах. Эта проверка границ выполняется

в методе ok(), который возвращает логическое значение true, если индекс

правильный, а иначе — логическое значение false.

А теперь рассмотрим следующий код аксессора set, предотвращающего ошибки

нарушения границ массива.

set

{

if (ok(index))

{

a[index] = value;

ErrFlag = false;

}

else ErrFlag = true;

}

Если параметр index метода ok() находится в установленных пределах, то соответствующему

элементу массива присваивается значение, передаваемое из параметра

value. В противном случае устанавливается логическое значение true переменной

ErrFlag. Напомним, что ***value*** в ***любом*** аксессорном методе является ***неявным параметром***,

***содержащим присваиваемое значение***. Его не нужно (да и нельзя) объявлять

отдельно.

Наличие обоих аксессоров, get и set, в индексаторе ***не является обязательным***.

Так, можно создать индексатор только для чтения, реализовав в нем один лишь аксессор

get, или же индексатор только для записи с единственным аксессором set.

# Перегрузка индексаторов

Индексатор может быть перегружен. В этом случае для выполнения выбирается

тот вариант индексатора, в котором точнее соблюдается соответствие его параметра и

аргумента, указываемого в качестве индекса. Ниже приведен пример программы, в которой

индексатор массива класса FailSoftArray перегружается для индексов типа

double. При этом индексатор типа double округляет свой индекс до ближайшего

целого значения.

(***glava10\_2***)

class FailSoftArray

{

int[] a; //link to base array

public int Length; //length of array opened

public bool ErrFlag; //result of last operation

//make array set size

public FailSoftArray(int size)

{

a = new int[size];

Length = size;

}

//indexer for this class

public int this[int index]

{

//accessor get

get

{

if (ok(index))

{

ErrFlag = false;

return a[index];

}

else

{

ErrFlag = true;

return 0;

}

}

//accessor set

set

{

if (ok(index))

{

a[index] = value;

ErrFlag = false;

}

else ErrFlag = true;

}

}

//another indexator

public int this[double idx]

{

//accessor get

get

{

int index;

//to integer

if ((idx - (int)idx) < 0.5) index = (int)idx;

else index = (int)idx + 1;

if (ok(index))

{

ErrFlag = false;

return a[index];

}

else

{

ErrFlag = true;

return 0;

}

}

//accessor set

set

{

int index;

if ((idx - (int)idx < 0.5)) index = (int)idx;

else index = (int)idx + 1;

if (ok(index))

{

a[index] = value;

ErrFlag = false;

}

else ErrFlag = true;

}

}

//return true, if index is in borders

private bool ok(int index)

{

if (index >= 0 & index < Length) return true;

return false;

}

}

class FSDemo

{

static void Main()

{

FailSoftArray fs = new FailSoftArray(5);

int x;

//put values in fs

for (int i = 0; i < fs.Length; i++)

fs[i] = i;

//now use indexes int and double

Console.WriteLine("fs[1] : " + fs[1]);

Console.WriteLine("fs[2] : " + fs[2]);

Console.WriteLine("fs[1.1] : " + fs[1.1]);

Console.WriteLine("fs[1.6] : " + fs[1.6]);

}

}

Как показывает приведенный выше результат, индексы типа double округляются

до ближайшего целого значения. В частности, индекс 1.1 округляется до 1, а индекс

1.6 — до 2. Программа не устанавливает значения, она лишь обращается по индексу, а индексатор как бы исправляет ошибку округляя до целого, хз, вот такой вот блять пример который путает.

Представленный выше пример программы наглядно демонстрирует правомочность

перегрузки индексаторов, но на практике она применяется нечасто. Как правило,

индексаторы перегружаются для того, чтобы использовать объект определенного

класса в качестве индекса, вычисляемого каким-то особым образом.

# Индексаторы без базового массива

Следует особо подчеркнуть, что индексатор совсем не обязательно должен оперировать

массивом. Его основное назначение — предоставить пользователю функциональные

возможности, аналогичные массиву. В качестве примера в приведенной ниже

программе демонстрируется индексатор, выполняющий роль массива только для чтения,

содержащего степени числа 2 от 0 до 15. Обратите внимание на то, что в этой

программе отсутствует конкретный массив. Вместо этого индексатор просто вычисляет

подходящее значение для заданного индекса.

(***glava10\_3***)

class PwrofTwo

{

//acces to logic array, it contains

//degree of 2 from 0 to 15

public int this[int index]

{

//calculate and return degree of 2

get

{

if ((index >= 0) && (index < 16)) return pwr(index);

else return -1;

}

//no set accessor

}

int pwr(int p)

{

int result = 1;

for (int i = 0; i < p; i++)

result \*= 2;

return result;

}

}

class UsePwrOfTwo

{

static void Main()

{

PwrofTwo pwr = new PwrofTwo();

Console.Write("First 8 degree of 2: ");

for (int i = 0; i < 8; i++)

Console.Write(pwr[i] + " ");

Console.WriteLine();

Console.Write("And here is some bugs: ");

Console.Write(pwr[-1] + " " + pwr[17]);

Console.WriteLine();

}

}

Обратите внимание на то, что в индексатор класса PwrOfTwo включен только аксессор

get, но в нем отсутствует аксессор set. Как пояснялось выше, такой индексатор

служит только для чтения. Следовательно, объект класса PwrOfTwo может указываться

только в правой части оператора присваивания, но не в левой его части. Например,

попытка ввести следующую строку кода в приведенную выше программу не приведет

к желаемому результату.

***pwr [0] = 11; // не подлежит компиляции***

Такой оператор присваивания станет причиной появления ошибки во время компиляции,

поскольку для индексатора не определен аксессор set.

На применение индексаторов накладываются два существенных ограничения.

Во-первых, значение, выдаваемое индексатором, нельзя передавать методу в качестве

параметра ref или out, поскольку в индексаторе не определено место в памяти для

его хранения. И во-вторых, индексатор должен быть членом своего класса и поэтому

не может быть объявлен как static.

# Многомерные индексаторы

Индексаторы можно создавать и для многомерных массивов. В качестве примера

ниже приведен двумерный отказоустойчивый массив. Обратите особое внимание на

объявление индексатора в этом примере.

(***glava10\_4***)

class FailSoftArray2D

{

int[,] a; //link on base two - dimensional

int rows, cols; //size of array

public int Length; //oppened var length of array

public bool ErrFlag; //result of last operation

//build array typed size

public FailSoftArray2D(int r, int c)

{

rows = r;

cols = c;

a = new int[rows, cols];

Length = rows \* cols;

}

//indexator for class

public int this[int index1, int index2]

{

//get accessor

get

{

if(ok(index1, index2))

{

ErrFlag = false;

return a[index1, index2];

}

else

{

ErrFlag = true;

return 0;

}

}

//set accessor

set

{

if (ok(index1, index2))

{

a[index1, index2] = value;

ErrFlag = false;

}

else ErrFlag = true;

}

}

//true if index in borders

private bool ok(int index1, int index2)

{

if (index1 >= 0 & index1 < rows

& index2 >= 0 & index2 < cols)

return true;

return false;

}

}

//show two dimension indexator

class TwoIndexerDemo

{

static void Main()

{

FailSoftArray2D fs = new FailSoftArray2D(3, 5);

int x;

//find erros

Console.WriteLine("Hidden bug.");

for (int i = 0; i < 6; i++)

fs[i, i] = i \* 10;

for (int i = 0; i < 6; i++)

{

x = fs[i, i];

if (x != -1) Console.Write(x + " ");

}

Console.WriteLine();

//Now show the error

Console.WriteLine("\nWith bugs.");

for (int i = 0; i < 6; i++)

{

fs[i, i] = i \* 10;

if(fs.ErrFlag)

Console.WriteLine("fs[" + i + ", " + i + "] outside of borders.");

}

for (int i = 0; i < 6; i++)

{

x = fs[i, i];

if (!fs.ErrFlag) Console.Write(x + " ");

else Console.WriteLine("fs[" + i + ", " + i + "] outside of borders.");

}

}

}

# Свойства

Еще одной разновидностью члена класса является *свойство.* Как правило, свойство

сочетает в себе поле с методами доступа к нему. Как было показано в приведенных

ранее примерах программ, поле зачастую создается, чтобы стать доступным для пользователей

объекта, но при этом желательно сохранить управление над операциями,

разрешенными для этого поля, например, ограничить диапазон значений, присваиваемых

данному полю. Этой цели можно, конечно, добиться и с помощью закрытой

переменной, а также методов доступа к ее значению, но свойство предоставляет более

совершенный и рациональный путь для достижения той же самой цели.

Свойства очень похожи на индексаторы. В частности, свойство состоит из имени и

аксессоров get и set. Аксессоры служат для получения и установки значения переменной.

Главное преимущество свойства заключается в том, что его имя может быть

использовано в выражениях и операторах присваивания аналогично имени обычной

переменной, но в действительности при обращении к свойству по имени автоматически

вызываются его аксессоры get и set. Аналогичным образом используются аксессоры

get и set индексатора.

Ниже приведена общая форма свойства:

**тип имя {**

**get {**

**// код аксессора для чтения из поля**

**}**

**set {**

**// код аксессора для записи в поле**

**}**

где *тип* обозначает конкретный тип свойства, например int, а *имя —* присваиваемое

свойству имя. Как только свойство будет определено, любое обращение к свойству по

имени приведет к автоматическому вызову соответствующего аксессора. Кроме того,

аксессор set принимает неявный параметр value, который содержит значение, присваиваемое

свойству.

Ниже приведен простой пример программы, в которой определяется свойство

MyProp, предназначенное для доступа к полю prop. В данном примере свойство допускает

присваивание только положительных значений.

(***glava10\_5***)

class SimpProp

{

int prop; //field, used by option MyProp

public SimpProp() { prop = 0; }

//this option can access to private var prop

//it checks if number is positive

public int MyProp

{

get

{

return prop;

}

set

{

if (value >= 0) prop = value;

}

}

}

class PropertyDemo

{

static void Main()

{

SimpProp ob = new SimpProp();

Console.WriteLine("First value of ob.MyProp: " + ob.MyProp);

ob.MyProp = 100; //assing var

Console.WriteLine("Current value ob.MyProp: " + ob.MyProp);

//cant assign with negative num

Console.WriteLine("Try to assign - 10");

ob.MyProp = -10;

Console.WriteLine("Curent value ob.MyProp: " + ob.MyProp);

}

}

Присваивание разрешено благодаря индексатору и ***set***.

В этом коде определяется одно закрытое поле prop и свойство MyProp, управляющее доступом к полю prop.

Как пояснялось выше, само свойство не определяет место в памяти для хранения поля,

а только управляет доступом к полю. Кроме того, поле prop является закрытым, а значит,

оно доступно *только* через свойство MyProp.

Аксессор get этого свойства просто возвращает значение из поля prop, тогда как аксессор ***set*** устанавливает значение в поле prop в том и только в том случае, если это значение оказывается положительным.

Тем не менее свойства можно

создавать доступными только для чтения или только для записи. Так, если требуется

создать свойство, доступное только для чтения, то достаточно определить единственный

аксессор get. А если нужно создать свойство, доступное только для записи, то

достаточно определить единственный аксессор set.

# Автоматически реализуемые свойства

Начиная с версии C# 3.0, появилась возможность для реализации очень простых

свойств, не прибегая к явному определению переменной, которой управляет свойство.

Вместо этого базовую переменную для свойства автоматически предоставляет компилятор.

Такое свойство называется *автоматически реализуемым* и принимает следующую

общую форму:

**тип имя { get; set; }**

где *тип* обозначает конкретный тип свойства, а *имя —* присваиваемое свойству имя.

Обратите внимание на то, что после обозначений аксессоров get и set сразу же следует

точка с запятой, а тело у них отсутствует.

Такой синтаксис предписывает компилятору создать автоматически переменную, иногда еще называемую *поддерживающим полем*, для хранения значения.

Ниже приведен пример объявления свойства, автоматически реализуемого под

именем UserCount.

***public int UserCount { get; set; }***

Как видите, в этой строке кода переменная явно не объявляется. И как пояснялось

выше, компилятор автоматически создает анонимное поле, в котором хранится значение.

А в остальном автоматически реализуемое свойство UserCount подобно всем

остальным свойствам.

Но в отличие от обычных свойств автоматически реализуемое свойство не может

быть доступным только для чтения или только для записи. При объявлении этого свойства

в любом случае необходимо указывать оба аксессора — get и set. Хотя добиться

желаемого (т.е. сделать автоматически реализуемое свойство доступным только для

чтения или только для записи) все же можно, объявив ненужный аксессор как ***private***

# Применение инициализаторов объектов в свойствах

Как пояснялось в главе 8, *инициализатор объекта* применяется в качестве альтернативы

явному вызову конструктора при создании объекта. С помощью инициализаторов

объектов задаются начальные значения полей или свойств, которые требуется инициализировать.

При этом синтаксис инициализаторов объектов оказывается одинаковым

как для свойств, так и для полей. В качестве примера ниже приведена программа

из главы 8, измененная с целью продемонстрировать применение инициализаторов

объектов в свойствах.

class MyClass

{

//now its options

public int Count { get; set; }

public string Str { get; set; }

}

class ObjInitDemo

{

static void Main()

{

//make obj with init

MyClass obj =

new MyClass { Count = 100, Str = "Test" };

Console.WriteLine(obj.Count + " " + obj.Str);

}

}

Как видите, свойства Count и Str устанавливаются в выражениях с инициализатором

объекта.

Как пояснялось в главе 8, синтаксис инициализатора объекта оказывается наиболее

пригодным для работы с анонимными типами, формируемыми в LINQ-выражениях.

А в остальных случаях чаще всего используется синтаксис обычных конструкторов.

# Ограничения, присущие свойствам

Свойствам присущ ряд существенных ограничений. Во-первых, свойство не определяет

место для хранения данных, и поэтому не может быть передано методу в качестве

параметра ref или out. Во-вторых, свойство не подлежит перегрузке. Наличие двух

разных свойств с доступом к одной и той же переменной допускается, но это, скорее,

исключение, чем правило. И наконец, свойство не должно изменять состояние базовой

переменной при вызове аксессора get. И хотя это ограничительное правило не

соблюдается компилятором, его нарушение считается семантической ошибкой. Действие

аксессора get не должно носить характер вмешательства в функционирование

переменной.

# Применение модификаторов доступа в аксессорах

По умолчанию доступность аксессоров set и get оказывается такой же, как и у

индексатора и свойства, частью которых они являются. Так, если свойство объявляется

как public, то по умолчанию его аксессоры set и get также становятся открытыми

(public). Тем не менее для аксессора set или get можно указать собственный модификатор

доступа, например private. Но в любом случае доступность аксессора,

определяемая таким модификатором, должна быть более ограниченной, чем доступность,

указываемая для его свойства или индексатора.

Существует целый ряд причин, по которым требуется ограничить доступность аксессора.

Допустим, что требуется предоставить свободный доступ к значению свойства,

но вместе с тем дать возможность устанавливать это свойство только членам его класса.

Для этого достаточно объявить аксессор данного свойства как private. В приведенном

ниже примере используется свойство MyProp, аксессор set которого указан как

(***glava10\_6***)

class PropAccess

{

int prop;

public PropAccess() { prop = 0; }

//this option has access to closed variable prop

//it grants to get value of prop from any code,

//but set only - members of class

public int MyProp

{

get

{

return prop;

}

private set //now its closed accessor

{

prop = value;

}

}

//this method increments the value of MyProp

public void IncrProp()

{

MyProp++; //only in class

}

}

//show acces of private set

class PropAccessDemo

{

static void Main()

{

PropAccess ob = new PropAccess();

Console.WriteLine("Value of ob.MyProp: " + ob.MyProp);

//ob.MyProp = 10; !Error bcs of private set

ob.IncrProp();

Console.WriteLine("Value of ob.MyProp: " + ob.MyProp);

}

}

В классе PropAccess аксессор set указан как private. Это означает, что он доступен

только другим членам данного класса, например методу IncrProp(), но недоступен

для кода за пределами класса PropAccess. Именно поэтому попытка Присвоить

свойству ob.МуРrор значение в классе PropAccessDemo закомментирована.

Вероятно, ограничение доступа к аксессорам оказывается наиболее важным для

работы с автоматически реализуемыми свойствами. Как пояснялось выше, создать

автоматически реализуемое свойство только для чтения или же только для записи

нельзя, поскольку оба аксессора, get и set, должны быть указаны при объявлении

такого свойства. Тем не менее добиться желаемого результата все же можно, объявив

один из аксессоров автоматически реализуемого свойства как private.

В качестве

примера ниже приведено объявление автоматически реализуемого свойства Length

для класса FailSoftArray, которое фактически становится доступным только для

чтения.

***public int Length { get; private set; }***

Свойство Length может быть установлено только из кода в его классе, поскольку

его аксессор set объявлен как private. А изменять свойство Length за пределами

его класса не разрешается. Это означает, что за пределами своего класса свойство, по

существу, оказывается доступным только для чтения.

Для опробования автоматически реализуемых вариантов свойств Length и Error

в классе FailSoftArray удалим сначала переменные len и ErrFlag, поскольку они

больше не нужны, а затем заменим каждое применение переменных len и ErrFlag

свойствами Length и Error в классе FailSoftArray. Ниже приведен обновленный

вариант класса FailSoftArray вместе с методом Main(), демонстрирующим его

применение.

(***glava10\_6.1***)

class FailSoftArray

{

int[] a; //link to array

//set array

public FailSoftArray(int size)

{

a = new int[size];

Length = size;

}

//Auto implemented and only for reading option Length

public int Length { get; private set; }

//Auto implemented and only for reading option Error

public bool Error { get; private set; }

//indexator for array

public int this[int index]

{

get

{

if (ok(index))

{

Error = false;

return a[index];

}

else

{

Error = true;

return 0;

}

}

set

{

if (ok(index))

{

a[index] = value;

Error = false;

}

else Error = true;

}

}

//true if index in borders

private bool ok(int index)

{

if (index >= 0 & index < Length) return true;

return false;

}

}

class FinalFSDemo

{

static void Main()

{

FailSoftArray fs = new FailSoftArray(5);

//Use option Error

for (int i = 0; i < fs.Length + 1; i++)

{

fs[i] = i \* 10;

if (fs.Error)

Console.WriteLine("Index Error " + i);

}

}

}

Этот вариант класса FailSoftArray действует таким же образом, как и предыдущий,

но в нем отсутствуют поддерживающие поля, объявляемые явно.

На применение модификаторов доступа в аксессорах накладываются следующие

ограничения. Во-первых, действию модификатора доступа подлежит только один аксессор:

set или get, но не оба сразу. Во-вторых, модификатор должен обеспечивать

более ограниченный доступ к аксессору, чем доступ на уровне свойства или индексатора.

И наконец, модификатор доступа нельзя использовать при объявлении аксессора

в интерфейсе или же при реализации аксессора, указываемого в интерфейсе.

# Применение индексаторов и свойств

В предыдущих примерах программ был продемонстрирован основной принцип

действия индексаторов и свойств, но их возможности не были раскрыты в полную

силу. Поэтому в завершение этой главы обратимся к примеру класса RangeArray,

в котором индексаторы и свойства используются для создания типа массива с пределами

индексирования, определяемыми пользователем.

Как вам должно быть уже известно, индексирование всех массивов в C# начинается

с нуля. Но в некоторых приложениях индексирование массива удобнее начинать с любой

произвольной точки отсчета: с 1 или даже с отрицательного числа, например от

-5 и до 5. Рассматриваемый здесь класс RangeArray разработан таким образом, чтобы

допускать подобного рода индексирование массивов.

Используя класс RangeArray, можно написать следующий фрагмент кода.

RangeArray rа = new RangeArray(-5, 10); // массив с индексами от -5 до 10

for(int i=-5; i <= 10; i++) ra[i] = i; // индексирование массива от -5 до 10

Нетрудно догадаться, что в первой строке этого кода конструируется объект класса

RangeArray с пределами индексирования массива от -5 до 10 включительно. Первый

аргумент обозначает начальный индекс, а второй — конечный индекс. Как только

объект rа будет сконструирован, он может быть проиндексирован как массив в пределах

от -5 до 10.

(***glava10\_7***)

class RangeArray

{

//closed data

int[] a;

int lowerBound;

int upperBound;

//auto implementer only for read option Length

public int Length { get; private set; }

//auto implementer only for read option Error

public bool Error { get; private set; }

//build array with size

public RangeArray(int low, int high)

{

high++;

if (high <= low)

{

Console.WriteLine("Wrong indexes");

high = 1; //create min array

low = 0;

}

a = new int[high - low];

Length = high - low;

lowerBound = low;

upperBound = --high;

}

//indexator

public int this[int index]

{

get

{

if (ok(index))

{

Error = false;

return a[index - lowerBound];

}

else

{

Error = true;

return 0;

}

}

set

{

if (ok(index))

{

a[index - lowerBound] = value;

Error = false;

}

else Error = true;

}

}

//true if in borders

private bool ok(int index)

{

if (index >= lowerBound & index <= upperBound) return true;

return false;

}

}

//use class

class RangeArrayDemo

{

static void Main()

{

RangeArray ra = new RangeArray(-5, 5);

RangeArray ra2 = new RangeArray(1, 10);

RangeArray ra3 = new RangeArray(-20, -12);

//use ra as array

Console.WriteLine("Lenght of ra: " + ra.Length);

for (int i = -5; i <= 5; i++)

ra[i] = i;

Console.Write("Content if ra: ");

for (int i = -5; i <= 5; i++)

Console.Write(ra[i] + " ");

Console.WriteLine();

//use ra2 as array

Console.WriteLine("Lenght of ra2: " + ra2.Length);

for (int i = 1; i <= 10; i++)

ra2[i] = i;

Console.Write("Content of ra2: ");

for (int i = 1; i <= 10; i++)

Console.Write(ra2[i] + " ");

Console.WriteLine();

//use ra3 as array

Console.WriteLine("Lenght of ra3: " + ra3.Length);

for (int i = -20; i <= -12; i++)

ra3[i] = i;

Console.Write("Content if ra3: ");

for (int i = -20; i <= -12; i++)

Console.Write(ra3[i] + " ");

Console.WriteLine();

}

}

Ниже приведен конструктор класса RangeArray.

public RangeArray(int low, int high)

{

high++;

if (high <= low)

{

Console.WriteLine("Wrong indexes");

high = 1; //create min array

low = 0;

}

a = new int[high - low];

Length = high - low;

lowerBound = low;

upperBound = --high;

}

При конструировании объекту класса RangeArray передается нижняя граница

массива в качестве параметра low, а верхняя граница — в качестве параметра high.

Затем значение параметра high инкрементируется, поскольку пределы индексирования

массива изменяются от low до high включительно. Далее выполняется следующая

проверка: является ли верхний индекс больше нижнего индекса. Если это не так, то

выдается сообщение об ошибке и создается массив, состоящий из одного элемента.

После этого для массива распределяется память, а ссылка на него присваивается переменной

а. Затем свойство Length устанавливается равным числу элементов массива.

И наконец, устанавливаются переменные lowerBound и upperBound.

Далее в классе RangeArray реализуется его индексатор, как показано ниже.

public int this[int index]

{

get

{

if (ok(index))

{

Error = false;

return a[index - lowerBound];

}

else

{

Error = true;

return 0;

}

}

set

{

if (ok(index))

{

a[index - lowerBound] = value;

Error = false;

}

else Error = true;

}

}

Этот индексатор подобен тому, что использовался в классе FailSoftArray, за одним

существенным исключением. Обратите внимание на следующее выражение, в котором

индексируется массив а.

a[index - lowerBound];

В этом выражении индекс, передаваемый в качестве параметра index, преобразуется

в индекс с отсчетом от нуля, пригодный для индексирования массива а. Данное

выражение действует при любом значении переменной lowerBound: положительном,

отрицательном или нулевом. Таким образом индекс ***a*** всегда будет равен 0.

Класс RangeArray демонстрирует лишь одну разновидность специализированного

массива, который может быть создан с помощью индексаторов и свойств. Существуют,

конечно, и другие. Аналогичным образом можно, например, создать динамические

массивы, которые расширяются или сужаются по мере надобности, ассоциативные

и разреженные массивы. Попробуйте создать один из таких массивов в качестве

упражнения.