## **Свойства**

 Свойство — это второй специальный тип членов класса. Свойство включает поле и методы доступа к этому полю. Часто требуется создать поле, которое должно быть доступно для пользователей объекта, но программист при этом хочет осуществлять управление операциями, разрешенными к выполнению над этим полем. Например, по некоторым обстоятельствам вы желаете ограничить диапазон значений, которые можно присваивать этому полю. Несмотря на то что этого можно достичь с помощью закрытой переменной и методов доступа к ней, свойство предлагает более удобный и простой способ решения этой задачи.

Свойства во многом напоминают индексаторы. Свойство состоит из имени и пары аксессоров (get и set). Аксессоры используются для чтения содержимого переменной и записи в нее нового значения. Основное достоинство свойства состоит в том, что его имя можно использовать в выражениях и инструкциях присваивания подобно обычной переменной, хотя в действительности здесь будут автоматически вызываться get- и set-аксессоры. Автоматический вызов аксессоров и роднит свойства с индексаторами. Формат записи свойства таков:

**тип имя{**

 **get{**

 **// код аксессора чтения поля**

 **}**

 **set{**

 **// код аксессора записи поля**

 **}**

Здесь тип — это тип свойства (например, int), а имя — его имя. После определения свойства любое использование его имени означает вызов соответствующего аксессора. Аксессор set автоматически принимает параметр с именем value, который содержит значение, присваиваемое свойству.

Важно понимать, что свойства не определяют область памяти. Следовательно, свойство управляет доступом к полю, но самого поля не обеспечивает. Это поле должно быть задано независимо от свойства.

Рассмотрим простой пример, в котором определяется свойство myprop, используемое для доступа к полю prop. Это свойство позволяет присваивать полю только положительные числа.

// Пример использования свойства.

 using System;

class SimpProp {

 int prop; // Это поле управляется свойством myprop.

 public SimpProp() {

 prop =0;

 }

 /\* Это свойство поддерживает доступ к закрытой

 переменной экземпляра prop. Оно позволяет

 присваивать ей только положительные числа. \*/

 public int myprop {

 get {

 return prop;

 }

 set {

 if(value >= 0)

 prop = value;

 }

 }

}

 // Демонстрируем использование свойства.

class PropertyDemo {

 public static void Main() {

 SimpProp ob = new SimpProp();

 Console.WriteLine("Исходное значение ob.myprop: " +

 ob.myprop);

 ob.myprop = 100; // Присваиваем значение.

 Console.WriteLine("Значение ob.myprop: " +

 ob.myprop);

 // Переменной prop невозможно присвоить

 // отрицательное значение.

 Console.WriteLine(

 "Попытка присвоить -10 свойству ob.myprop");

 ob.myprop = -10;

 Console.WriteLine("Значение ob.myprop: " + ob.myprop);

 }

}

Результаты выполнения этой программы выглядят так:

Исходное значение ob.myprop: 0

Значение ob.myprop: 100

Попытка присвоить -10 свойству ob.myprop

Значение ob.myprop: 100

На этой программе стоит остановиться подробнее. В классе SimpProp определяется закрытое поле prop и свойство myprop, которое управляет доступом к полю prop. Как упоминалось выше, свойство само не определяет область хранения поля, а лишь управляет доступом к нему. Поэтому без определения базового поля определение свойства теряет всякий смысл. Более того, поскольку поле prop закрытое, к нему можно получить доступ только посредством свойства myprop. Свойство myprop определяется как public-член класса, чтобы к нему можно было обратиться с помощью кода вне класса, включающего это свойство. В этом есть своя логика, поскольку свойство предоставляет доступ к закрытому полю prop с помощью аксессоров: get-аксессор просто возвращает значение prop, a set-аксессор устанавливает новое значение prop, если оно положительно. Таким образом, свойство myprop управляет тем, какие значения может содержать поле prop. В этом и состоит важность свойств.

Свойство myprop предназначено для чтения и записи, поскольку позволяет как прочитать содержимое своего базового поля, так и записать в него новое значение. Но можно создавать свойства, предназначенные только для чтения (определив лишь get-аксессор) либо только для записи (определив лишь set-аксессор). Мы можем использовать свойство для дальнейшего усовершенствования класса, определяющего отказоустойчивый массив. Как вы уже знаете, с каждым массивом связано свойство Length. До сих пор в классе FailSoftArray для этой цели просто использовалось открытое целочисленное поле Length. Такое решение — не самое лучшее, поскольку в этом случае можно записать в поле Length значение, отличное от реальной длины этого массива. (Например, какой-нибудь программист с дурными наклонностями мог умышленно ввести это значение.) Потенциально опасную ситуацию можно исправить, заменив открытую переменную Length свойством, предназначенным только для чтения, как показано в следующей версии класса FailSoftArray.

// Добавляем в класс FailSoftArray свойство Length.

 using System;

 class FailSoftArray {

 int[] a; // Ссылка на базовый массив.

 int len; // Длина массива, основа для свойства Length.

 public bool errflag; // Индикатор результата

 // последней операции.

 // Создаем массив заданного размера.

 public FailSoftArray(int size) {

 a = new int[size];

 len = size;

 }

 // Свойство Length предназначено только для чтения.

 public int Length {

 get {

 return len;

 }

 }

 // Это — индексатор класса FailSoftArray.

 public int this[int index] {

 // Это — get-аксессор.

 get {

 if(ok(index)) {

 errflag = false;

 return a[index];

 } else {

 errflag = true;

 return 0;

 }

 }

 // Это — set-аксессор.

 set {

 if(ok(index)) {

 a[index] = value;

errflag = false;

 }

 else errflag = true;

 }

 }

 // Метод возвращает true, если индекс внутри границ.

 private bool ok(int index) {

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

 return false;

 }

}

 // Демонстрируем улучшенный отказоустойчивый массив.

class ImprovedFSDemo {

 public static void Main() {

 FailSoftArray fs = new FailSoftArray(5);

 int x;

 // Свойство Length можно считывать.

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

 fs[i] = i\*10;

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

 x = fs[i];

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

 }

 Console.WriteLine();

 // fs.Length = 10; // Ошибка, запись запрещена!

 }

}

Теперь Length — это свойство, которое в качествве области памяти использует закрытую переменную len. В этом свойстве определен только get-аксессор, и потому свойство Length можно только читать, но не изменять. Чтобы убедиться в этом, попробуйте убрать символы комментариев в начале следующей строки программы:

 // fs.Length = 10; // Ошибка, запись запрещена!

При попытке скомпилировать программу с этой строкой кода вы получите сообщение об ошибке, уведомляющее о том, что свойство Length предназначено только для чтения. Внесение в класс FailSoftArray свойства Length значительно улучшило его, но на этом рано ставить точку. Член errflag — еще один кандидат на “превращение” из обычной переменной экземпляра в свойство со всеми преимуществами, поскольку доступ к нему следует ограничить до определения “только для чтения”. Приводим окончательную версию класса FailSoftArray, в которой создано свойство Error, использующее в качестве области хранения индикатора ошибки исходную переменную errflag.

// Превращаем переменную errflag в свойство.

 using System;

 class FailSoftArray {

 int[] a; // Ссылка на базовый массив.

 int len; // Длина массива.

bool errflag; // Теперь этот член закрыт.

 // Создаем массив заданного размера.

 public FailSoftArray(int size) {

 a = new int[size];

 len = size;

 }

 // Свойство Length предназначено только для чтения.

 public int Length {

 get {

 return len;

 }

 }

 // Свойство Error предназначено только для чтения.

 public bool Error {

 get {

 return errflag;

 }

 }

 // Это - индексатор класса FailSoftArray.

 public int this[int index] {

 // Это — get-аксессор.

 get {

 if(ok(index)) {

 errflag = false;

 return a[index];

 } else {

 errflag = true;

 return 0;

 }

 }

 // Это — set-аксессор.

 set {

 if(ok(index)) {

 a[index] = value;

 errflag = false;

 }

 else

 errflag = true;

 }

 }

 // Метод возвращает true, если индекс внутри границ.

 private bool ok(int index) {

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

 return false;

 }

}

// Демонстрируем улучшенный отказоустойчивый массив.

class FinalFSDemo {

 public static void Main() {

 FailSoftArray fs = new FailSoftArray(5);

 // Используем свойство Error.

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

 fs[i] = i\*10;

 if(fs.Error)

 Console.WriteLine("Ошибка в индексе " + i);

 }

 }

}

Создание свойства Error заставило нас внести в класс FailSoftArray два изменения. Во-первых, переменную errflag пришлось сделать закрытой, поскольку она теперь используется в качестве базовой области памяти для свойства Error. В результате к члену errflag теперь нельзя обращаться напрямую. Во-вторых, добавлено свойство Error, предназначенное только для чтения. Отныне программы для выявления ошибок будут опрашивать не поле errflag, а свойство Error. Это продемонстрировано в методе Main(), в котором умышленно генерируется ошибка нарушения границ массива, а для ее обнаружения используется свойство Error.

***Правила использования свойств***

На использование свойств налагаются довольно серьезные ограничения. Во-первых, поскольку в свойстве не определяется область памяти, его нельзя передавать методу в качестве ref- или out-параметра. Во-вторых, свойство нельзя перегружать. (Но при необходимости вы можете иметь два различных свойства, которые используют одну и ту же базовую переменную, но к такой организации свойств прибегают нечасто.) Наконец, свойство не должно изменять состояние базовой переменной при вызове get-аксессора, хотя несоблюдение этого правила компилятор обнаружить не в состоянии. Другими словами, get-операция должна быть максимально простой.

 ***Использование индексаторов и свойств***

Несмотря на то что в предыдущих примерах продемонстрирован механизм работы индексаторов и свойств, в них не отражена в полной мере вся мощь этих программных средств. В заключение этой главы мы рассмотрим класс RangeArray, использующий индексаторы и свойства для создания массива такого типа, в котором индексный диапазон массива определяется программистом.

Как вы знаете, в C# индексация массивов начинается с нуля. Однако в некоторых приложениях было бы удобно начинать индексацию массивов с произвольного числа, например с единицы или даже с отрицательного числа, чтобы индексы изменялись в диапазоне от -5 до 5. Приведенный здесь класс RangeArray как раз и позволяет подобные способы индексации массивов. При использовании класса RangeArray можно написать такие строки кода:

RangeArray rа = new RangeArray(-5, 10); // Массив с

 // индексами от -5 до 10.

 for(int i = -5; i <= 10; i++) ra[i] = i; // Индекс

 // изменяется от -5 до 10.

Нетрудно догадаться, что первая строка создает объект класса RangeArray (массив ra), в котором индексы изменяются от -5 до 10 включительно. Первый аргумент задает начальный индекс, а второй — конечный. Ниже представлены классы RangeArray и RangeArrayDemo, демонстрирующие использование этого массива. Класс RangeArray поддерживает массив int-элементов, но при необходимости можно заменить этот тип данных другим.

/\* Создание класса для поддержки массива с заданным диапазоном индексации. Класс RangeArray позволяет начинать индексацию с числа, отличного от нуля. При создании объекта класса RangeArray необходимо задать индексы начала и конца диапазона. Отрицательные индексы также допустимы. Например, можно создать массивы с диапазоном изменения индексов от -5 до 5, от 1 до 10 или от 50 до 56. \*/

using System;

class RangeArray {

 // Закрытые данные.

 int[] a; // Ссылка на базовый массив.

 int lowerBound; // Наименьший индекс.

 int upperBound; // Наибольший индекс.

 // Данные для свойств.

 int len; // Базовая переменная для свойства Length.

 bool errflag; // Базовая переменная для свойства Error.

 // Создаем массив с заданным размером.

 public RangeArray(int low, int high) {

 high++;

 if(high <= low) {

 Console.WriteLine("Неверные индексы.");

 high = 1; // Создаем минимальный массив для

 // безопасности.

 low = 0;

 }

 a = new int[high - low];

 len = high - low;

 lowerBound = low;

 upperBound = --high;

 }

 // Свойство Length, предназначенное только для чтения.

 public int Length {

 get {

 return len;

 }

 }

 // Свойство Error, предназначенное только для чтения.

 public bool Error {

 get {

 return errflag;

 }

 }

 // Это — индексатор для класса RangeArray.

 public int this[int index] {

// Это — get-аксессор.

 get {

 if(ok(index)) {

 errflag = false;

 return a[index - lowerBound];

 } else {

 errflag = true;

 return 0;

 }

 }

 // Это — set-аксессор.

 set {

 if(ok(index)) {

 a[index - lowerBound] = value;

 errflag = false;

 }

 else errflag = true;

 }

 }

 // Метод возвращает true, если индекс находится

 // внутри границ.

 private bool ok(int index) {

 if(index >= lowerBound & index <= upperBound)

 return true;

 return false;

 }

}

 // Демонстрируем использование массива с произвольно

// заданным диапазоном индексов.

class RangeArrayDemo {

 public static void Main() {

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

 RangeArray ra2 = new RangeArray(1, 10);

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

 // Используем массив ra.

 Console.WriteLine("Длина массива ra: " + ra.Length);

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

 ra[i] = i;

 Console.Write("Содержимое массива ra: ");

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

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

 Console.WriteLine("\n");

 // Используем массив ra2,

 Console.WriteLine("Длина массива ra2: " + ra2.Length);

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

 ra2[i] = i;

 Console.Write("Содержимое массива ra2: ");

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

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

 Console.WriteLine("\n");

 // Используем массив ra3

 Console.WriteLine("Длина массива ra3: " + ra3.Length);

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

 ra3[i] = i;

 Console.Write("Содержимое массива ra3: ");

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

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

 Console.WriteLine("\n");

 }

}

При выполнении эта программа генерирует такие результаты:

Длина массива ra: 11

Содержимое массива ra: -5 -4 -3 -2 -1 0 1 2 3 4 5

Длина массива ra2: 10

Содержимое массива rа2: 1 2 3 4 56 7 8 9 10

Длина массива ra3: 9

Содержимое массива ra3: -20 -19 -18 -17 -16 -15 -14 -13 -12

Как подтверждают результаты выполнения этой программы, объекты типа RangeArray могут быть индексированы не обязательно начиная с нуля. Рассмотрим, как же реализован класс RangeArray.

Определение этого класса начинается с определения закрытых переменных экземпляра:

 // Закрытые данные.

 int[] а; // Ссылка на базовый массив.

 int lowerBound; // Наименьший индекс.

 int upperBound; // Наибольший индекс.

 // Данные для свойств.

 int len; // Базовая переменная для свойства Length.

 bool errflag; // Базовая переменная для свойства Error.

Базовый массив имеет имя а. Он размешается в памяти с помощью конструктора класса RangeArray. Индекс нижней границы массива сохраняется в закрытой переменной lowerBound, а индекс верхней границы — в закрытой переменной upperBound. Затем объявляются переменные элемента, которые поддерживают свойства Length и Error. Конструктор класса RangeArray имеет такой вид:

// Создаем массив с заданным размером.

public RangeArray(int low, int high) {

 high++;

 if(high <= low) {

 Console.WriteLine("Неверные индексы.");

 high = 1; // Создаем минимальный массив для

 // безопасности.

 low = 0;

}

 а = new int[high - low];

 len = high - low;

 lowerBound = low;

 upperBound = --high;

}

Объект типа RangeArray создается в результате передачи нижнего граничного индекса в параметр low и верхнего граничного индекса — в параметр high. Значение high затем инкрементируется, чтобы вычислить размер массива, поскольку задаваемые индексы изменяются от low до high включительно. После этого проверяем, действительно ли верхний индекс больше нижнего. Если это не так, выводится сообщение об ошибке в индексах и создается одноэлементный массив. Затем выделяется область памяти для массива (либо корректно заданного, либо ошибочно), и ссылка на эту область присваивается переменной а. А переменная len (на которой основано свойство Length) устанавливается равной количеству элементов в массиве. Наконец, устанавливаются закрытые переменные lowerBound и upperBound.

В классе RangeArray затем реализуются свойства Length и Error. Ниже приведены их определения.

// Свойство Length, предназначенное только для чтения.

public int Length {

 get {

 return len;

 }

}

// Свойство Error, предназначенное только для чтения.

public bool Error {

 get {

 return errflag;

 }

}

Эти свойства аналогичны свойствам, используемым классом FailSoftArray, и их работа организована подобным образом. Далее в классе RangeArray реализуется индексатор. Вот его определение:

// Это — индексатор для класса RangeArray.

public int this[int index] {

 // Это — get-аксессор.

 get {

 if(ok(index)) {

 errflag = false;

 return a[index - lowerBound];

 } else {

 errflag = true;

 return 0;

 }

 }

 // Это — set-аксессор.

 set {

 if(ok(index)) {

 a[index - lowerBound] = value;

 errflag = false;

 } else

 errflag = true;

}

}

Этот индексатор очень похож на индексатор класса FailSoftArray, но с одним важным отличием. Обратите внимание на выражение, которое служит в качестве значения индекса массива а.

 index - lowerBound

Это выражение преобразует реальный индекс, переданный через параметр index, в “нормальный”, т.е. в значение, которое имел бы индекс текущего элемента, если бы индексирование массива начиналась с нуля. Ведь только такое индексирование подходит для базового массива а. Это выражение работает при любом значении переменной lowerBound: положительном, отрицательном или нулевом.

Теперь осталось рассмотреть определение метода ok().

// Метод возвращает значение true, если индекс

// находится внутри границ.

private bool ok(int index) {

 if(index >= lowerBound & index <= upperBound)

 return true;

 return false;

}

Это определение аналогично тому, которое используется в классе FailSoftArray за исключением того, что попадание индекса в нужный диапазон проверяется с помощью значений переменных lowerBound и upperBound.

Класс RangeArray иллюстрирует только один вид массива, создаваемого “под заказ” с помощью индексаторов и свойств. Можно также создавать динамические массивы (которые расширяются и сокращаются по необходимости), ассоциативные и разреженные массивы.