Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Заведующий  методическим кабинетом  \_\_\_\_\_\_\_\_\_\_\_\_\_Е.В.Фалей  « \_\_\_\_ » \_\_\_\_\_\_\_\_\_ 2017 |

|  |  |
| --- | --- |
| Специальность: 2-40 01 01 «Программное обеспечение информационных технологий» | Дисциплина: «Учебная практика по программированию» |

**Практическая работа №16**

**Инструкционно-технологическая карта**

Тема:Обобщения.

* Обзор обобщений. Роль параметров обобщенных типов (2 часа).
* Создание обобщенного метода. Обобщенные делегаты (2 часа).
* Создание обобщенных структур и классов. Обобщенные интерфейсы (2 часа).

Цель: Сформировать навыки работы с обобщениями. Научиться создавать обобщенные методы, делегаты, структуры, классы, интерфейсы.

Время выполнения: 6 часов

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическими сведениями.
2. Выполнить рассмотренные в теоретическом материале примеры.
3. Получить у преподавателя вариант задания и выполнить его.
4. Оформить выполненное задание в отчет.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

**Обзор обобщений**

Чтобы понять, что такое обобщения, стоит присмотреться к задаче, для решения которой они были разработаны. Предположим, нужно смоделировать структуру наподобие очереди, работающей по принципу «первым пришел — первым ушел». Для этого можно создать следующий класс:

class Queue {

private const int DEFAULTQUEUESIZE = 100;

private int[] data;

private int head = 0, tail = 0;

private int numElements = 0;

public Queue()

{

this.data = new int[DEFAULTQUEUESIZE];

}

public Queue(int size)

{

if (size > 0)

{

this.data = new int[size];

}

else

{

throw new ArgumentOutOfRangeException("size", "Must be greater than zero");

}

}

public void Enqueue(int item)

{

if (this.numElements == this.data.Length)

{

throw new Exception("Queue full");

}

this.data[this.head] = item;

this.head++;

this.head %= this.data.Length;

this.numElements++;

}

public int Dequeue()

{

if (this.numElements == 0)

{

throw new Exception("Queue empty");

}

int queueItem = this.data[this.tail];

this.tail++;

this.tail %= this.data.Length;

this.numElements--;

return queueItem;

}

}

Для создания кольцевого буфера хранения данных в этом классе используется массив. Размер этого массива указан конструктором. Чтобы добавить элемент в очередь, приложение использует метод Enqueue, а чтобы удалить его из оче­реди — метод Dequeue. Закрытые поля head и tail отслеживают, куда нужно вставить элемент в массив и из какого места извлечь элемент из массива. Поле numElements показывает, сколько элементов находится в массиве. Методы Enqueue и Dequeue используют эти поля для определения того, где сохранять или откуда извлекать элемент, и выполняют ряд простейших проверок на наличие ошибок. Как показано в следующем примере кода, приложение может создать Queue-объект и вызвать эти методы. Обратите внимание на то, что элементы извлекаются из очереди в том же порядке, в котором в нее попадают:

Queue queue = new Queue(); // Создание нового Queue-объекта

queue.Enqueue(100);

queue.Enqueue(-25);

queue.Enqueue(33);

Console.WriteLine($"{queue.Dequeue()}"); // Выводит на экран 100

Console.WriteLine($"{queue.Dequeue()}"); // Выводит на экран -25

Console.WriteLine($"{queue.Dequeue()}"); // Выводит на экран 33

Класс Queue хорошо работает для очередей из int-элементов, а как быть, если нужно создавать очереди из строк, или чисел с плавающей точкой, или даже более сложных типов, таких как Circle, Horse или? Проблема в том, что способ реализации класса Queue ограничивает тип элементов целочисленными значениями, и при попытке вставить элемент Horse-типа будет выдана ошибка компиляции:

Queue queue = new Queue();

Horse myHorse = new Horse();

queue.Enqueue(myHorse); // Ошибка в ходе компиляции: конвертация Horse в int // невозможна

Один из способов обойти это ограничение заключается в указании того, что массив в классе Queue содержит элементы типа object, а также в изменении конструктора и методов Enqueue и Dequeue таким образом, чтобы они принимали параметр типа object и возвращали значение такого же типа:

class Queue {

private object[] data;

public Queue()

{

this.data = new object[DEFAULTQUEUESIZE];

}

public Queue(int size)

{

this.data = new object[size];

}

public void Enqueue(object item)

{

}

public object Dequeue()

{

object queueltem = this.data[this.tail];

return queueItem;

}

}

Вспомним, что для ссылки на значение или переменную любого типа можно воспользоваться типом object. Все ссылочные типы автоматически наследуются (как непосредственно, так и косвенно) из класса System.Object среды Microsoft .NET Framework (в C# object является псевдонимом System.Object). Теперь, по­скольку методы Enqueue и Dequeue распоряжаются объектами, в очередях можно работать с классами Circles, Horses, Whales или любыми другими, встречавшимися ранее в этой книге. Но важно учесть, что возвращаемые методом Dequeue значения следует приводить к соответствующему типу, поскольку компилятор не будет выполнять автоматическое преобразование из типа object:

Queue queue = new Queue();

Horse myHorse = new Horse();

queue.Enqueue(myHorse); // Теперь это допустимо, поскольку Horse - это объект

Horse dequeuedHorse = (Horse)queue.Dequeue(); // Следует привести тип obj'ect к типу Horse

Если не выполнить приведение типа возвращаемого значения, будет получена ошибка компиляции, сообщающая о невозможности неявного приведения типа object к типу Horse. Это требование явного приведения типа во многом снижает присущую типу object гибкость. Кроме того, совсем не трудно допустить ошибку, написав следующий код:

Queue queue = new Queue();

Horse myHorse = new Horse(); queue.Enqueue(myHorse);

Circle myCircle = (Circle)queue.Dequeue(); // Ошибка в ходе выполнения

Несмотря на то что этот код пройдет компиляцию, он не сможет работать и в ходе выполнения выдаст исключение неверного приведения типов System.InvalidCastException. Допущенная ошибка связана с попыткой при извлечении из очереди ссылки на Horse сохранить ее в Circle-переменной, а эти два типа не совместимы друг с другом. Эта ошибка ничем не проявится до тех пор, пока код не будет запущен на выполнение, поскольку у компилятора недостаточно информации для выполнения проверки в ходе компиляции. Каков реальный тип объекта, извлекаемого из очереди, выясняется только при выполнении кода.

Еще одним недостатком использования подхода с применением значений типа object для создания обобщенных классов и методов является вероятность дополнительного потребления памяти и процессорного времени, если в ходе выполнения потребуется преобразовывать объект в значение типа и обратно. Рассмотрим следующий фрагмент кода, в котором ведется работа с очередью из целочисленных значений:

Queue queue = new Queue(); int myint = 99;

queue.Enqueue(mylnt); // Запаковка int в object

myInt = (int)queue.Dequeue(); // Распаковка object в int

Тип данных Queue предполагает, что содержащиеся в нем элементы имеют тип object, а он относится к ссылочным типам. Помещение в очередь элемента, относящегося к типам значений, например, int-элемента, требует его упаковки с целью преобразования в ссылочный тип. Аналогично этому извлечение из очереди элемента с получением int-значения требует распаковки элемента с целью его преобразования в элемент, имеющий тип значения. Хотя упаковка и распаковка происходят без каких-либо дополнительных действий, они вызывают дополнительные издержки производительности, поскольку используют динамическое выделение памяти. Для отдельно взятого элемента эти издержки невелики, но когда программа создает очереди из большого количества элементов, имеющих тип значения, они существенно возрастают.

**Роль параметров обобщенных типов**

Чтобы избавиться от необходимости приведения типов, в C# предоставляются обобщения, сокращающие объем требуемых упаковок и упрощающие создание обобщенных классов и методов, которые принимают параметры типа, указывающие типы объектов, с которыми они работают. В C# указанием на то, что класс является обобщением, служит параметр типа, заключенный в угловые скобки:

class Queue<T>

{

}

Параметр типа T в этом примере работает как заполнитель для реального типа, применяемого в ходе компиляции. При написании кода для получения экземпляра класса-обобщения Queue вы предоставляете тип, который нужно подставить вместо T (Circle, Horse, int и т. д.). При определении в классе полей и методов для указания их типов используется точно такой же заполнитель:

class Queue<T>

{

private T[] data; // Массив из элементов типа 'T', где 'T' - параметр типа

public Queue()

{

this.data = new T[DEFAULTQUEUESIZE]; // Использование 'T' в качестве типа

// данных

}

public Queue(int size)

{

this.data = new T[size];

}

public void Enqueue(T item) // Использование 'T' в качестве типа параметра метода

{

}

public T Dequeue() // Использование 'T' в качестве типа возвращаемого значения

{

T queueltem = this.data[this.tail]; // Данные в массиве относятся к типу 'T'

return queueItem;

}

}

В качестве параметра типа T может использоваться любой допустимый в C# идентификатор, хотя чаще всего это одиночный символ «T». Вместо него ставится тип, указанный при создании Queue-объекта. В следующем примере создается Queue-объект из int-значений и Queue-объект из Horse-значений:

Queue<int> intQueue = new Queue<int>();

Queue<Horse> horseQueue = new Queue<Horse>();

Кроме этого, у компилятора теперь достаточно информации для строгой проверки типов при сборке приложения. Теперь больше не нужно приводить типы данных при вызове метода Dequeue, и компилятор может отлавливать ошибки несоответствия типов на ранней стадии:

intQueue.Enqueue(99);

int myInt = intQueue.Dequeue(); // Приведение типов не требуется

Horse myHorse = intQueue.Dequeue(); // Ошибка компилятора: выполнить неявное

// преобразование типа 'int' в тип 'Horse' невозможно

Следует иметь в виду, что такая подстановка конкретного типа вместо T не является простым механизмом текстового замещения. Компилятор выполняет полноценную семантическую подстановку, поэтому для T можно указать любой допустимый тип. Рассмотрим еще несколько примеров:

struct Person {

}

Queue<int> intQueue = new Queue<int>();

Queue<Person> personQueue = new Queue<Person>();

В первом примере создается очередь из целочисленных значений, а во втором — из Person-значений. Для каждой очереди компилятор создает также версии методов Enqueue и Dequeue. Для очереди intQueue эти методы имеют следующий вид:

public void Enqueue(int item); public int Dequeue();

А для очереди personQueue они имеют следующий вид:

public void Enqueue(Person item); public Person Dequeue();

Сравните эти определения с теми упомянутыми в предыдущем разделе версиями класса Queue, которые основаны на применении типа object. В методах, происходящих из класса-обобщения, параметр item передается методу Enqueue в виде значения, относящегося к типу значений, не требующих упаковки. Аналогично этому значение, возвращенное методом Dequeue, относится к типу значений, не нуждающихся в распаковке. Аналогичный набор методов создается и для двух других очередей.

Параметр типа не обязательно должен быть простым классом или принадлежать к типу значений. Можно, к примеру, создать очередь из очередей целочисленных значений (если, конечно, она когда-нибудь пригодится):

Queue<Queue<int>> queueQueue = new Queue<Queue<int>>();

У класса-обобщения может быть несколько параметров типа. Например, класс-обобщение Dictionary, определение которого находится в пространстве имен System.Collections.Generic библиотеки классов .NET Framework, предполагает использование двух параметров типа: одного для ключей, а другого для значений.

Примечание: Можно также, используя точно такой же синтаксис с параметрами типа, как и для обобщенных классов, объявлять обобщенные структуры и интерфейсы.

**Сравнение классов-обобщений и обобщенных классов**

Следует различать класс-обобщение, использующий параметры типа, и обоб­щенный класс, разработанный с целью получения параметров, которые могут приводиться к различным типам. Например, показанная ранее версия класса Queue, основанная на применении значений типа object, является обобщенным классом. Это обособленная реализация класса, и его методы получают параметры типа object, а также возвращают значения типа object. Этот класс можно использовать с целочисленными, строковыми и многими другими типами, но всякий раз используется экземпляр одного и того же класса и применяемые данные приходится приводить к object-типу и из object-типа.

Сравните это с классом Queue<T>. При каждом использовании этого класса с параметром типа (например, Queue<int> или Queue<Horse>) вы заставляете компилятор создавать абсолютно новый класс с функциональными возмож­ностями, определяемыми классом-обобщением. Это означает, что Queue<int> по типу совершенно отличается от Queue<Horse>, одинаковым у них является только поведение. Класс-обобщение можно представить себе как определение шаблона, который затем по мере надобности используется компилятором для создания новых классов с конкретным указанием типов. Версии класса-обоб­щения с конкретным указанием типов (Queue<int>, Queue<Horse> и т. д.) называются классами со сконструированными типами, и их можно рассматривать как совершенно разные типы (даже те, у которых один и тот же набор методов и свойств).

**Обобщения и ограничения**

Временами нужно будет, чтобы параметр типа, используемый классом-обобщением, идентифицировал тип, предоставляющий конкретные методы. Например, если определяется класс PrintableCollection (коллекция, пригодная для вывода на печать), могут понадобиться гарантии того, что все объекты, хранящиеся в классе, имели метод Print. Это условие можно задать, использовав ограничение.

Используя ограничение, можно ограничить параметры типа класса-обобщения теми типами, которые реализуют конкретный набор интерфейсов, и тем самым предоставлять методы, определенные этими интерфейсами. Например, если в интерфейсе IPrintable определяется метод Print, вы можете создать следующий класс PrintableCollection:

public class PrintableCollection<T> where T : IPrintable

При создании этого класса с параметром типа компилятор выполняет проверку, чтобы убедиться, что тип, используемый для T, действительно реализует интерфейс IPrintable. Если этот интерфейс в нем не реализуется, компиляция прекращает работу с выдачей ошибки.

**Создание обобщенных структур и классов**

В пространстве имен System.Collections.Generic библиотеки классов .NET Framework содержится целый ряд готовых к использованию классов-обобщений. Можно также определять собственные классы-обобщения, чем вы и будете за­ниматься в данном разделе. Но прежде чем приступить к их созданию, давайте рассмотрим некоторые теоретические положения.

В следующих упражнениях будет определяться и использоваться класс, представляющий собой двоичное дерево, которое является полезной структурой данных, используемой для выполнения разнообразных операций, включая очень быструю сортировку и сквозной поиск данных.

Двоичное дерево является рекурсивной (автореферентной) структурой данных, которая может быть пустой или содержать три элемента: единицу информации (datum), которую обычно называют узлом, и два поддерева, которые также являются двоичными деревьями. Два поддерева традиционно называются левым и правым, поскольку обычно их изображают соответственно левее и правее узла. Каждое левое или правое поддерево является либо пустым, либо содержащим узел и другие поддеревья. Теоретически структура может быть бесконечной. Небольшое двоичное дерево показано на рисунке 1

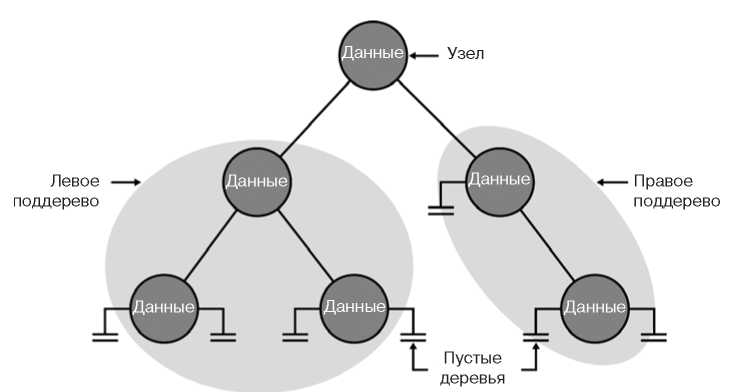


Рисунок 1

Реальная эффективность двоичных деревьев проявляется при их использовании для сортировки данных. При наличии изначально неупорядоченной последовательности объектов одного типа можно выстроить упорядоченное двоичное дерево, а затем перемещаться по нему, заходя в каждый узел упорядоченной последовательности. Далее показан алгоритм вставки элемента «I» в упорядоченное двоичное дерево «B»:

Если дерево "В" пустое,

То

Создать новое дерево "В" с новым элементом "I" в качестве узла и

пустыми левым и правым поддеревьями

Иначе

Исследовать значения текущего узла "N" данного дерева "B"

Если значение "N" больше, чем значение нового элемента "I",

То

Если левое поддерево "B" пустое,

То

Создать новое левое поддерево дерева "B" с элементом "I" в качестве

узла и пустыми левым и правым поддеревьями

Иначе

Вставить "I" в левое поддерево дерева "B"

Завершить условие Если

Иначе

Если правое поддерево дерева "В" пустое

То

Создать новое правое поддерево дерева "B" с элементом "I" в качестве

узла и пустыми левым и правым поддеревьями

Иначе

Вставить "I" в правое поддерево дерева "B"

Завершить условие Если

Завершить условие Если

Завершить условие Если

Обратите внимание на рекурсивность этого алгоритма, вызывающего самого себя для вставки элемента в левое или правое поддерево в зависимости от результата сравнения значения элемента со значением текущего узла в дереве.

Примечание: Определение выражения «больше чем» зависит от типа данных в элементе и в узле. Для числовых данных «больше чем» может быть простым арифметическим сравнением, а для текстовых данных оно может быть сравнением строк, но другим типам данных нужно давать их собственные средства сравнения значений.

Если изначально имеются пустое двоичное дерево и неупорядоченная последовательность объектов, можно устроить сквозной обход этой последовательности, вставляя каждый объект в двоичное дерево с использованием данного алгоритма и получив в результате упорядоченное дерево. Этапы построения дерева из набора, состоящего из пяти целых чисел, показаны на рисунке 2

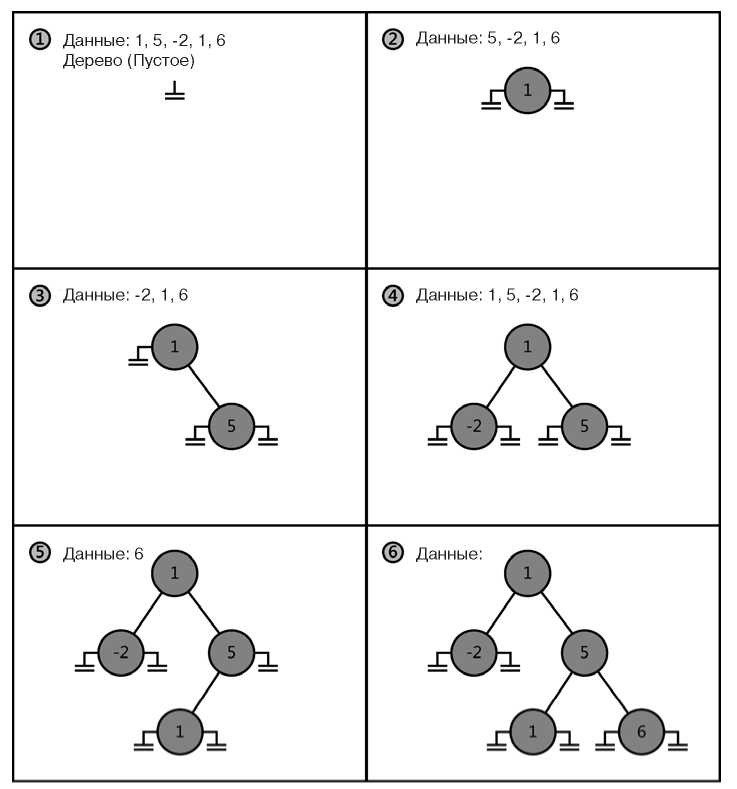


Рисунок 2

После построения упорядоченного двоичного дерева его содержимое можно показать в виде последовательности путем поочередного посещения каждого узла и вывода на экран найденного значения. Алгоритм выполнения этой задачи также имеет рекурсивную природу:

Если левое поддерево не пустое

То

Вывести содержимое левого поддерева

Завершить условие Если

Вывести значение узла

Если правое поддерево не пустое

То

Вывести содержимое правого поддерева

Завершить условие Если

Этапы выведения содержимого дерева на экран показаны на рисунке 3. Обратите внимание на то, что теперь целые числа выводятся на экран в порядке их возрастания.

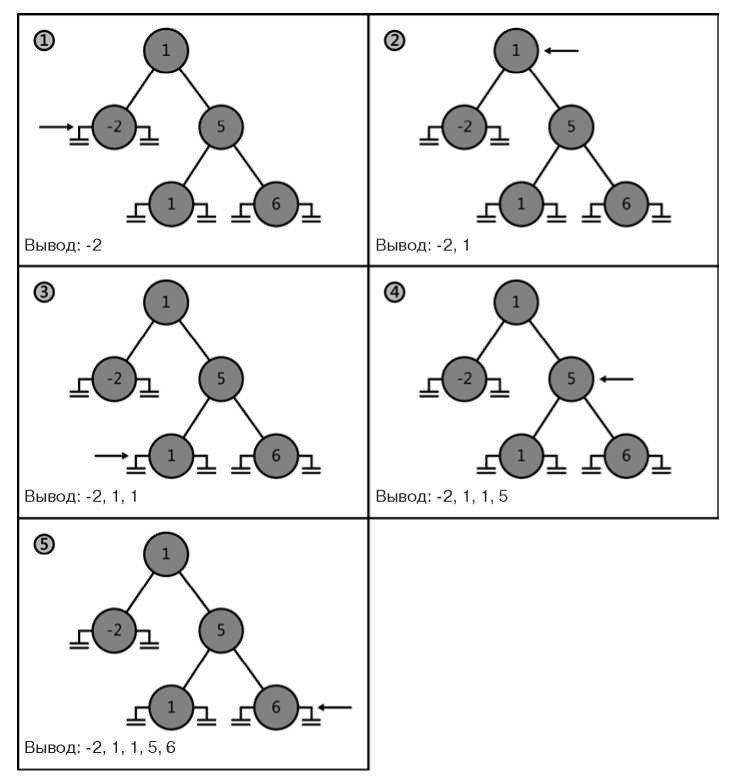


Рисунок 3

**Создание класса двоичного дерева с использованием обобщений**

В следующем упражнении обобщения будут использованы для определения класса двоичного дерева, способного содержать почти любой тип данных. Единственное ограничение заключается в том, что тип данных должен предоставлять средства сравнения значений различных экземпляров.

Класс двоичного дерева может пригодиться в множестве различных приложений. Поэтому вы будете создавать его не в виде самостоятельного приложения, а в качестве библиотеки классов. Затем этот класс можно будет использовать где-нибудь в другом месте, при этом не понадобятся копирование исходного кода и его повторная компиляция. Библиотека классов представляет собой набор откомпилированных классов (и других типов, таких как структуры и делегаты), сохраненный в сборке. Сборка является файлом, имя которого обычно имеет расширение .dll. Элементы библиотеки классов могут использоваться другими проектами и приложениями посредством добавления ссылки на их сборку с последующим перенесением их пространств имен в область видимости путем использования директив using. Именно это вы и сделаете при тестировании класса двоичного дерева.

Алгоритм вставки узла в двоичное дерево требует от вас сравнения значения вставляемого узла со значениями уже имеющихся в дереве узлов. Если используется числовой тип, например int, можно воспользоваться операторами <, > и ==. Но как сравнивать объекты, если используется какой-нибудь другой тип, например, Mammal или Circle?

Если нужно создать класс, требующий от вас возможности сравнения значений в соответствии с неким естественным (а может, и неестественным) упорядоче­нием, следует реализовать интерфейс IComparable. Этот интерфейс содержит метод CompareTo, принимающий один параметр, указывающий на объект, срав­ниваемый с текущим экземпляром, и возвращающий целое число, показывающее результат сравнения.

|  |  |
| --- | --- |
| Значение | Означает |
| Меньше 0 | Текущий экземпляр меньше значения параметра |
| 0 | Текущий экземпляр равен значению параметра |
| Больше 0 | Текущий экземпляр больше значения параметра |

Рассмотрим в качестве примера класс Circle:

class Circle

{

public Circle(int initialRadius)

{

radius = initialRadius;

}

public double Area()

{

return Math.PI \* radius \* radius;

}

private double radius;

}

Сделать класс пригодным для сравнения можно, реализовав интерфейс System.IComparable и предоставив метод CompareTo. В данном примере метод CompareTo сравнивает Circle-объекты на основе их площадей. Круг с большей площадью считается больше круга с меньшей площадью:

class Circle : System.IComparable

{

public int CompareTo(object obj)

{

Circle circObj = (Circle)obj; // Приведение параметра к его

// настоящему типу if (this.Area() == circObj.Area()) return 0;

if (this.Area() > circObj.Area()) return 1;

return -1;

}

}

Если изучить интерфейс System.IComparable, можно заметить, что его параметр определен как object. Но этот подход не дает полной независимости от применяемых типов. Чтобы понять, почему так, представьте, что получится, если попытаться передать методу CompareTo что-нибудь не относящееся к типу Circle. Интерфейс System.IComparable требует для доступа к методу Area использования приведения типа. Если параметр не относится к типу Circle и является объектом какого-либо другого типа, приведение даст сбой. Но в пространстве имен System определяется также интерфейс-обобщение IComparable<T>, в котором содер­жится следующий метод:

int CompareTo(T other);

Обратите внимание на то, что этот метод принимает не значение типа object, а параметр типа (T) и поэтому он намного безопаснее той версии интерфейса, которая не использует обобщение. Следующий код показывает, как этот интер­фейс можно реализовать в классе Circle:

class Circle : System.IComparable<Circle>

{

public int CompareTo(Circle other)

{

if (this.Area() == other.Area()) return 0;

if (this.Area() > other.Area()) return 1;

return -1;

}

}

Параметр для метода CompareTo должен соответствовать типу, указанному в интерфейсе IComparable<Circle>. В целом предпочтительнее реализовывать интерфейс System.IComparable<T>, а не интерфейс System.IComparable. Можно также реализовать оба интерфейса, что и делают большинство типов в среде .NET Framework.

Создание класса Tree<TItem>

Укажите в меню Файл среды Microsoft Visual Studio 2015 на пункт Создать и щелкните на пункте Проект. В левой панели Шаблоны появившегося на экране диалогового окна Создание проекта щелкните на пункте Visual C#. Выберите в средней панели шаблон Библиотека классов. Наберите в поле Имя строку BinaryTree. Укажите в поле Расположение папку \Microsoft Press\VCSBS\Chapter 17 вашей папки документов, а затем щелкните на кнопке OK.

Щелкните в обозревателе решений правой кнопкой мыши на файле Class1.cs, щелкните на пункте Переименовать, а затем измените имя файла на Tree.cs. При появлении предложения позвольте среде Visual Studio изменить имя класса, а также имя файла.

Измените в окне редактора определение класса Tree на Tree<TItem>, как показано в следующем коде жирным шрифтом:

public class Tree<TItem>

{

}

Измените в окне редактора определение класса Tree<Tltem> таким образом, чтобы указать, что тип параметра Tltem должен обозначать тип, в котором реализуется интерфейс-обобщение IComparable<TItem>. Необходимые изменения выделены в следующем примере кода жирным шрифтом. Измененное определение класса Tree<TItem> должно приобрести следующий вид:

public class Tree<TItem> where TItem : IComparable<TItem>

{

}

Добавьте к классу Tree<Tltem> три открытых автоматически создаваемых свойства: TItem-свойство по имени NodeData и Tree<TItem>-свойства с именами LeftTree и RightTree, выделенные в следующем примере кода жирным шрифтом:

public class Tree<TItem> where TItem : IComparable<TItem>

{

public TItem NodeData { get; set; }

public Tree<TItem> LeftTree { get; set; }

public Tree<TItem> RightTree { get; set; }

}

Добавьте к классу Tree<TItem> конструктор, принимающий единственный Tltem-параметр по имени nodeValue. Установите в конструкторе для свойства NodeData значение nodeValue и инициализируйте свойства LeftTree и RightTree null-значениями, как показано в выделенном жирным шрифтом фрагменте следующего кода:

public class Tree<TItem> where TItem : IComparable<TItem>

{

public Tree(TItem nodeValue)

{

this.NodeData = nodeValue; this.LeftTree = null; this.RightTree = null;

}

}

ПРИМЕЧАНИЕ: Обратите внимание на то, что имя конструктора не включает параметр типа: конструктор называется Tree, а не Tree<TItem>.

Добавьте к классу Tree<TItem> открытый метод по имени Insert, выделенный в следующем коде жирным шрифтом. Этот метод вставляет TItem-значение в дерево.

Определение метода должно иметь следующий вид:

public class Tree<TItem> where TItem : IComparable<TItem>

{

public void Insert(TItem newItem)

{

}

}

Реализуйте в методе Insert рассмотренный ранее рекурсивный алгоритм для создания упорядоченного двоичного дерева. Конструктор создает исходный узел дерева, поэтому метод Insert может считать, что дерево не пустое. Следующий код является частью алгоритма, запускаемого после проверки дерева на пустоту. Он показан здесь, чтобы помочь разобраться в коде, который будет вами написан для метода Insert при реализации следующих этапов упражнения:

…

Изучить значение узла "N" дерева "В"

Если значение "N" больше значения нового элемента "I"

То

Если левое поддерево "B" пустое

То

Создать новое левое поддерево дерева "B" с элементом "I" в качестве узла, с пустыми левым и правым поддеревьями

Иначе

Вставить "I" в левое поддерево дерева "В"

Завершить условие Если

…

Добавьте к методу Insert инструкцию, объявляющую локальную переменную типа TItem по имени currentNodeValue. Инициализируйте эту переменную, как показано в следующем коде жирным шрифтом, значением свойства NodeData, принадлежащего дереву:

public void Insert(TItem newItem)

{

TItem currentNodeValue = this.NodeData;

}

Добавьте к методу Insert выделенную в следующем примере кода жирным шрифтом инструкцию if-else, поставив ее после определения переменной currentNodeValue. Эта инструкция использует метод CompareTo, определенный в интерфейсе IComparable<T>, чтобы выяснить, является ли значение текущего узла большим, чем значение нового элемента:

public void Insert(TItem newItem)

{

TItem currentNodeValue = this.NodeData;

if (currentNodeValue.CompareTo(newItem) > 0)

{

// Вставка нового элемента в левое поддерево

}

else

{

// Вставка нового элемента в правое поддерево

}

}

Добавьте к части кода if сразу же после комментария // Вставка нового элемента в левое поддерево следующие инструкции:

if (this.LeftTree == null)

{

this.LeftTree = new Tree<TItem>(newItem);

}

else

{

this.LeftTree.Insert(newItem);

}

Эти инструкции проверяют левое поддерево на пустоту. Если оно пустое, создается новое поддерево с использованием нового значения, которое прикрепляется к текущему узлу, в противном случае новое значение вставляется в существующее левое поддерево путем рекурсивного вызова метода Insert.

Добавьте к части кода else самой внешней инструкции if-else сразу же после комментария // Вставка нового элемента в правое поддерево следующий эквивалентный код, вставляющий новый узел в правое поддерево:

if (this.RightTree == null)

{

this.RightTree = new Tree<TItem>(newItem);

}

else

{

this.RightTree.Insert(newItem);

}

Добавьте к классу Tree<TItem> после метода Insert еще один открытый метод по имени WalkTree. Этот метод осуществляет сквозной обход дерева, посещая последовательно каждый узел и создавая строку, представляющую данные, содержащиеся в дереве. Определение метода должно иметь следующий вид:

public string WalkTree()

{

}

Добавьте к методу WalkTree инструкции, выделенные в следующем примере кода жирным шрифтом. Эти инструкции реализуют алгоритм, рассмотренный ранее и предназначенный для обхода двоичного дерева. При посещении каждого узла его значение возвращается методом в строку:

public string WalkTree()

{

string result = "";

if (this.LeftTree != null)

{

result = this.LeftTree.WalkTree();

}

result += $" {this.NodeData.ToString()} ";

if (this.RightTree != null)

{

result += this.RightTree.WalkTree();

}

return result;

}

Щелкните в меню Сборка на пункте Собрать решение. Класс должен без проблем пройти компиляцию, но при обнаружении ошибок исправьте код и выполните повторную сборку решения.

В следующем упражнении вы тестируете класс Tree<Tltem> путем создания двоичных деревьев из целых чисел и строк.

Тестирование класса Tree<TItem>

Щелкните правой кнопкой мыши в обозревателе решений на решении BinaryTree, укажите на пункт Добавить и щелкните на пункте Создать проект.

ПРИМЕЧАНИЕ: Щелкнуть правой кнопкой мыши нужно на решении BinaryTree, а не на проекте BinaryTree.

Добавьте новый проект, воспользовавшись шаблоном Консольное приложение. Присвойте проекту имя BinaryTreeTest. Укажите для его размещения папку \Microsoft Press\VCSBS\Chapter17 вашей папки документов, а затем щелкните на кнопке OK.

ПРИМЕЧАНИЕ: Решение среды Visual Studio 2015 может содержать более одного проекта. Используйте эту особенность для добавления к решению BinaryTree второго проекта для тестирования класса Tree<TItem>.

Щелкните в обозревателе решений правой кнопкой мыши на проекте BinaryTreeTest, а затем щелкните на пункте Назначить автозагружаемым проектом (Set As Startup Project). В обозревателе решений будет выделен проект BinaryTreeTest. При запуске приложения будет фактически выполняться именно этот проект.

Щелкните в обозревателе решений правой кнопкой мыши на проекте BinaryTreeTest, укажите на пункт Добавить, а затем щелкните на пункте Ссылка.

Раскройте в левой панели появившегося диалогового окна Менеджер ссылок — BinaryTreeTest пункт Проекты, а затем щелкните на пункте Решение. Выберите в средней панели проект BinaryTree, установив флажок слева от него, после чего щелкните на кнопке OK.

На этом этапе к списку ссылок для проекта BinaryTreeTest в обозревателе решений будет добавлена сборка BinaryTree. Если в обозревателе решений просмотреть папку Ссылки для проекта BinaryTreeTest, то можно увидеть сбоку BinaryTree, показанную в самом верху. Теперь вы сможете создавать в проекте BinaryTreeTest объекты Tree<TItem>.

ПРИМЕЧАНИЕ: Если проект библиотеки классов не является частью того же решения, что и использующий его проект, вы должны добавить ссылку на сборку (на файл с расширением .dll), а не на проект библиотеки классов. Это можно сделать путем поиска сборки в диалоговом окне менеджера ссылок.

Добавьте в окне редактора, показывающего класс Program, находящийся в файле Program.cs, следующую директиву using, поставив ее в список в самом начале класса:

using BinaryTree;

Добавьте к методу Main инструкции, выделенные в следующем примере кода жирным шрифтом:

static void Main(string[] args)

{

Tree<int> treel = new Tree<int>(10);

tree1.Insert(5);

treel.Insert(ll);

tree1.Insert(5);

tree1.Insert(-12);

tree1.Insert(15);

tree1.Insert(0);

tree1.Insert(14);

tree1.Insert(-8);

tree1.Insert(10);

tree1.Insert(8);

tree1.Insert(8);

string sortedData = tree1.WalkTree();

Console.WriteLine($”Sorted data is: {sortedData}”);

}

Эти инструкции создают новое двоичное дерево для хранения целочисленных значений. Конструктор создает исходный узел, содержащий значение 10. Инструкции Insert добавляют к дереву узлы, а метод WalkTree создает строку, показывающую содержимое дерева, которое при выводе строки на экран должно появиться отсортированным в возрастающем порядке.

ПРИМЕЧАНИЕ: Не забывайте, что ключевое слово int в C# является всего лишь псевдонимом для типа System.Int32: когда объявляется int-переменная, на самом деле объявляется структурная переменная типа System.Int32. Тип System.Int32 реализует интерфейсы IComparable и IComparable<T>, благодаря чему вы можете создавать Tree<int>-объекты. Аналогично этому ключевое слово string является псевдонимом для System.String, где также реализуются интерфейсы IComparable и IComparable<T>.

Щелкните в меню Сборка на пункте Собрать решение и добейтесь того, чтобы решение прошло компиляцию. При необходимости исправьте ошибки.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что программа запускается и выводит значения в такой последовательности:

-12 -8 0 5 5 8 8 10 10 11 14 15

Нажмите клавишу Ввод и вернитесь в среду Visual Studio 2015.

Добавьте к методу Main класса Program ниже имеющегося кода следующие инструкции, выделенные жирным шрифтом:

static void Main(string[] args)

{

...

Tree<string> tree2 = new Tree<string>("Hello");

tree2.Insert("World");

tree2.Insert("How");

tree2.Insert("Are");

tree2.Insert("You");

tree2.Insert("Today");

tree2.Insert("I");

tree2.Insert("Hope");

tree2.Insert("You");

tree2.Insert("Are");

tree2.Insert("Feeling");

tree2.Insert("Well");

tree2.Insert("!");

sortedData = tree2.WalkTree();

Console.WriteLine($"Sorted data is: {sortedData}");

}

Эти инструкции создают еще одно двоичное дерево для хранения строк, заполняют его тестовыми данными и выводят на экран. На этот раз данные сортируются в алфавитном порядке.

Щелкните в меню Сборка на пункте Собрать решение и добейтесь того, чтобы решение прошло компиляцию. При необходимости исправьте ошибки.

Щелкните в меню Отладка на пункте Запуск без отладки. Убедитесь в том, что программа запускается и выводит целочисленные значения, как и раньше, а за ними выводятся строковые значения в такой последовательности (рисунок 4):

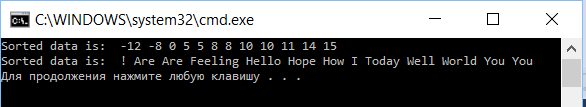


Рисунок 4

Нажмите клавишу Ввод и вернитесь в среду Visual Studio 2015.

**Создание обобщенного метода. Обобщенные делегаты**

Наряду с определением классов-обобщений можно создавать и методы-обобщения.

В методах-обобщениях можно указывать типы параметров и тип возвращаемого значения, используя параметр типа, который применяется практически так же, как и при определении класса-обобщения. Таким образом можно определить методы-обобщения, работа которых не зависит от типа, и избежать издержек на приведение типов (и в некоторых случаях на упаковку). Методы-обобщения зачастую используются в сочетании с классами-обобщениями. Вам они понадобятся как методы, получающие в качестве параметров обобщенные типы, или как методы, у которых тип возвращаемого значения обозначается обобщением.

Методы-обобщения определяются путем использования такого же синтаксиса параметра типа, который применялся при создании классов-обобщений (можно также указать ограничения). Например, метод-обобщение Swap<T>, показываемый в следующем фрагменте кода, выполняет для своих параметров обмен значениями. Поскольку эта функция полезна независимо от типа обмениваемых данных, есть смысл определить ее в виде метода-обобщения:

static void Swap<T>(ref T first, ref T second)

{

T temp = first;

first = second;

second = temp;

}

Метод вызывается указанием соответствующего для его параметра типа. В следующем примере показано, как вызвать метод Swap<T> для обмена двух целых чисел и двух строк:

int a = 1, b = 2;

Swap<int>(ref a, ref b);

string si = "Hello", s2 = "World";

Swap<string>(ref s1, ref s2);

ПРИМЕЧАНИЕ: Как создание экземпляров класса-обобщения с различными параметрами типа заставляет компилятор создавать различные типы, так и каждое отдельно взятое использование метода Swap<T> заставляет компилятор создавать различные версии метода. Метод Swap<int> не является эквивалентом метода Swap<string>, оба метода просто создаются из одного шаблона-обобщения, демонстрируя одинаковое поведение, но по отношению к различным типам.

**Определение метода-обобщения для создания двоичного дерева**

В предыдущем примере вы создали класс-обобщение для реализации двоичного дерева. Для добавления элементов данных в дерево в классе Tree<Tltem> предоставляется метод Insert. Но если потребуется добавить большое количество элементов, повторные вызова метода Insert создадут явные неудобства. В следующем упражнении вы создадите метод-обобщение InsertIntoTree, который можно будет использовать для вставки в дерево списка элементов, предоставленных за один вызов. Этот метод будет протестирован путем его использования для вставки списка символов в дерево символов.

**Написание метода InsertIntoTree**

Создайте в среде Visual Studio 2015 новый проект, воспользовавшись для этого шаблоном консольного приложения. В диалоговом окне Создание проекта введите для проекта имя BuildTree. Укажите для его размещения папку \Microsoft Press\VCSBS\Chapter17 вашей папки документов. В раскрывающемся списке Решение выберите пункт Создать новое решение, после чего щелкните на кнопке OK.

Щелкните в меню Проект на пункте Добавить ссылку. Щелкните в диалоговом окне Менеджер ссылок — BuildTree на кнопке Обзор (не перепутайте ее с одноименной вкладкой, находящейся в левой панели).

В диалоговом окне Выберите файлы, на которые нужно установить ссылки перейдите в папку \MicrosoftPress\VCSBS\Chapter17\BinaryTree\ BinaryTree\bin\Debug вашей папки документов, щелкните на файле BinaryTree.dll, после чего щелкните на кнопке Добавить.

Убедитесь в присутствии файла BinaryTree.dll в перечне сборок в диалоговом окне Менеджер ссылок — BuildTree и в том, что рядом с его именем установлен флажок, после чего щелкните на кнопке OK. Сборка BinaryTree будет добавлена к перечню ссылок, показанному в обозревателе решений.

Добавьте в окне редактора, показывающего содержимое файла Program.cs, в самое начало кода следующую директиву using:

using BinaryTree;

Следует напомнить, что класс Tree<Tltem> находится именно в этом пространстве имен.

Добавьте после метода Main класса Program метод InsertIntoTree. Он должен быть объявлен статическим и пустым (static void), получающим параметр Tree<TItem> и params-массив из элементов типа TItem по имени data. Параметр tree следует передать по ссылке, исходя из соображений, которые будут рассмотрены при выполнении следующего этапа упражнения.

Определение метода должно иметь следующий вид:

static void InsertIntoTree<TItem>(ref Tree<TItem> tree, params TItem[] data)

{

}

Тип TItem, который используется для элементов, вставляемых в двоичное дерево, должен реализовывать интерфейс IComparable<TItem>. Измените определение метода InsertIntoTree, добавив условие where, выделенное в следующем коде жирным шрифтом:

static void InsertIntoTree<TItem>(ref Tree<TItem> tree, params TItem[] data) where TItem : IComparable<TItem>

{

}

Добавьте к методу InsertIntoTree инструкции, выделенные в следующем примере кода жирным шрифтом. Эти инструкции выполняют сквозной обход списка params, добавляя каждый элемент к дереву путем использования метода Insert. Если указанное параметром tree значение изначально null, создается новый экземпляр Tree<TItem> — именно поэтому параметр tree передается по ссылке:

static void InsertIntoTree<TItem>(ref Tree<TItem> tree, params TItem[] data) where TItem : IComparable<TItem>

{

foreach (TItem datum in data)

{

if (tree == null)

{

tree = new Tree<TItem>(datum);

}

else

{

tree.Insert(datum);

}

}

}

**Тестирование метода InsertIntoTree**

Добавьте к методу Main класса Program следующие инструкции, выделенные жирным шрифтом, которые создают новый экземпляр Tree для хранения символьных данных, заполняя его проверочными данными путем использования метода InsertIntoTree, а затем выведите его на экран, воспользовавшись методом WalkTree класса Tree:

static void Main(string[] args)

{

Tree<char> charTree = null;

InsertIntoTree<char>(ref charTree, 'M', 'X', 'A', 'M', 'Z', 'Z', 'N');

string sortedData = charTree.WalkTree();

Console.WriteLine($”Sorted data is: {sortedData}”);

}

Щелкните в меню Сборка на пункте Собрать решение и добейтесь того, чтобы решение прошло компиляцию. При необходимости исправьте ошибки.

Щелкните в меню Отладка на пункте Запуск без отладки. Программа запустится и выведет символьные значения в следующем порядке:

A M M N X Z Z

Нажмите клавишу Ввод и вернитесь в среду Visual Studio 2015.

**Обобщенные интерфейсы**

**Вариантность и интерфейсы-обобщения**

Как известно, тип object может использоваться для хранения значения или ссылки на любой другой объект. Например, вполне допустимо воспользоваться следующим кодом:

string myString = "Hello";

object myObject = myString;

Вспомним, что в понятиях наследования класс String является производным класса Object, следовательно, все строки являются объектами.

Рассмотрим теперь следующий интерфейс и класс, которые являются обобщениями:

interface IWrapper<T>

{

void SetData(T data);

T GetData();

}

class Wrapper<T> : IWrapper<T>

{

private T storedData;

void IWrapper<T>.SetData(T data)

{

this.storedData = data;

}

T IWrapper<T>.GetData()

{

return this.storedData;

}

}

Класс Wrapper<T> предоставляет простую оболочку для указанного класса. Интерфейс IWrapper определяет метод SetData, который реализуется классом Wrapper<T> для сохранения данных, и метод GetData, который реализуется классом Wrapper<T> для извлечения данных. Вы можете создать экземпляр этого класса и воспользоваться им в качестве оболочки для строки:

Wrapper<string> stringWrapper = new Wrapper<string>();

IWrapper<string> storedStringWrapper = stringWrapper;

storedStringWrapper.SetData("Hello");

Console.WriteLine($"Stored value is {storedStringWrapper.GetDataQ}");

Этот код создает экземпляр типа Wrapper<string>. Чтобы вызвать метод SetData, он ссылается на объект через интерфейс IWrapper<string>. (В типе Wrapper<T> его интерфейсы реализуются явным образом, поэтому вам нужно вызывать методы через соответствующую ссылку на интерфейс.) Код также через интерфейс IWrapper<string> вызывает метод GetData. Если запустить этот код, он выведет на экран сообщение «Stored value is Hello» («Сохраненным значением является Hello»).

Рассмотрим следующую строку кода:

IWrapper<object> storedObjectWrapper = stringWrapper;

Имеющаяся в ней инструкция похожа на ту, что создавала ссылку на IWrapper<string> в предыдущем примере кода, — разница в том, что параметром типа является object, а не string. Может ли этот код считаться допустимым?

Вспомним, что все строки по сути являются объектами (вы можете, как показано ранее, присвоить строковое значение ссылке на объект), следовательно, теоретически эта инструкция вселяет надежды. Но если попробовать ее в деле, она не пройдет компиляцию, вызвав появление сообщения «Cannot implicitly convert type ‘Wrapper<string>’ to ‘IWrapper<object>’» («Неявное преобразование типа ‘Wrapper<string>’ в тип ‘IWrapper<object>’ выполнить невозможно»).

Можно попробовать применить явное приведение типа:

IWrapper<object> storedObjectWrapper = (IWrapper<object>)stringWrapper;

Это код пройдет компиляцию, но даст сбой в ходе выполнения приложения с выдачей исключения о недопустимом приведении типа — InvalidCastException. Проблема в том, что хотя все строки являются объектами, обратное утверждение неверно. Если бы выполнение этой инструкции было разрешено, то вы могли бы написать следующий код, который в конечном счете попытался бы сохранить Circle-объект в строковом поле:

IWrapper<object> storedObjectWrapper = (IWrapper<object>)stringWrapper;

Circle myCircle = new Circle();

storedObjectWrapper.SetData(myCircle);

Интерфейс IWrapper<T> называется инвариантным (неизменяемым). Вы не можете присвоить объект IWrapper<A> ссылке типа IWrapper<B>, даже если тип A является производным от типа B. В C# это ограничение реализуется по умолчанию, чтобы обеспечить безопасность вашего кода в отношении типов.

**Ковариантные интерфейсы**

Предположим, что вместо интерфейса IWrapper<T> вы определили интерфейсы IStoreWrapper<T> и IRetrieveWrapper<T>, показанные в следующем примере, и реализовали их в классе Wrapper<T> следующим образом:

interface IStoreWrapper<T>

{

void SetData(T data);

}

interface IRetrieveWrapper<T>

{

T GetData();

}

class Wrapper<T> : IStoreWrapper<T>, IRetrieveWrapper<T>

{

private T storedData;

void IStoreWrapper<T>.SetData(T data)

{

this.storedData = data;

}

T IRetrieveWrapper<T>.GetData()

{

return this.storedData;

}

}

Функционально класс Wrapper<T> остался таким же, как и прежде, за исключением того, что доступ к методам SetData и GetData осуществляется через разные интерфейсы:

Wrapper<string> stringWrapper = new Wrapper<string>();

IStoreWrapper<string> storedStringWrapper = stringWrapper;

storedStringWrapper.SetData("Hello");

IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper;

Console.WriteLine($"Stored value is {retrievedStringWrapper.GetDataQ}");

Тогда допустим ли следующий код?

IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;

Можно дать быстрый ответ, что нет, и код не пройдет компиляцию с указанием той же ошибки, что и прежде. Но если вдуматься, становится понятным, что хотя компилятор C# посчитал эту инструкцию небезопасной с точки зрения типов, причины такого решения уже нельзя считать обоснованными. Интерфейс IRetrieveWrapper<T> позволяет с помощью метода GetData всего лишь считывать данные, хранящиеся в Wrapper<T>-объекте, и не предоставляет какого-либо способа изменения данных. В подобных ситуациях, когда в интерфейсе-обобщении параметр типа фигурирует только для возвращаемых методами значений, вы можете проинформировать компилятор, что неявные преобразования вполне допустимы и что он не должен принуждать вас к строгому соблюдению без­опасности в отношении типов. Такое информирование осуществляется при объявлении параметра типа с помощью указания ключевого слова out:

interface IRetrieveWrapper<out T>

{

T GetData();

}

Эта особенность называется ковариантностью. При условии допустимости преобразования из типа A в тип B или при условии, что тип A является производным от типа B, вы можете присвоить IRetrieveWrapper<A>-объект IRetrieveWrapper<B>-ссылке. Следующий код теперь проходит компиляцию и выполняется вполне ожидаемым образом:

// тип string является производным от типа object, поэтому теперь код является

// вполне допустимым

IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;

Квалификатор out можно указать с параметром типа, только если параметр типа фигурирует исключительно для типа возвращаемых методами значений. Если параметр типа используется для указания типа любого из параметров метода, применение квалификатора out недопустимо и ваш код не пройдет компиляцию. К тому же ковариантность работает только со ссылочными типами. Дело в том, что типы значений не могут создавать иерархии наследования. Следовательно, следующий код не пройдет компиляцию, поскольку int относится к типу значений:

Wrapper<int> intWrapper = new Wrapper<int>();

IStoreWrapper<int> storedIntWrapper = intWrapper; // Эта инструкция допустима

// Следующая инструкция недопустима - целочисленные значения не являются объектами

IRetrieveWrapper<object> retrievedObjectWrapper = intWrapper;

Ковариантность демонстрируют несколько интерфейсов, определенных в .NET Framework, включая интерфейс IEnumerable<T>.

ПРИМЕЧАНИЕ: Ковариантными могут объявляться только интерфейсы и делегаты. Для классов-обобщений модификатор out не указывается.

**Контравариантные интерфейсы**

Контравариантность следует тем же принципам, что и ковариантность, за исключением того, что она работает в противоположном направлении, позволяя использовать интерфейс-обобщение для ссылки на объект типа B через ссылку на тип A при условии, что тип B является производным от типа A. Сразу усвоить это нелегко, поэтому лучше посмотрим на пример из библиотеки классов .NET Framework.

Пространство имен System.Collections.Generic в .NET предоставляет интерфейс под названием IComparer, имеющий следующий вид:

public interface IComparer<in T>

{

int Compare(T x, T y);

}

В классе, реализующем этот интерфейс, должен быть определен метод Compare, используемый для сравнения двух объектов того типа, который указан параметром типа T. Ожидается, что метод Compare будет возвращать целочисленное значение: нуль, если у параметров x и у одинаковые значения, отрицательное число, если значение x меньше значения у, и положительное число, если значение x больше значения у. Следующий код показывает пример, где объекты сортируются в соответствии с их хэш-кодом. (Метод GetHashCode реализован классом Object. Он просто возвращает целое число, идентифицирующее объект. Все ссылочные типы наследуют этот метод и могут его перегрузить своими собственными реализациями.)

class ObjectComparer : IComparer<Object>

{

int IComparer<Object>.Compare(Object x, Object y)

{

int xHash = x.GetHashCode(); int yHash = y.GetHashCode();

if (xHash == yHash) return 0;

if (xHash < yHash) return -1;

return 1;

}

}

Для сравнения двух объектов можно создать ObjectComparer-объект и через интерфейс IComparer<Object> вызвать метод Compare:

Object x = ...;

Object y = ...;

ObjectComparer objectComparer = new ObjectComparer();

IComparer<Object> objectComparator = objectComparer;

int result = objectComparator.Compare(x, y);

Выглядит скучновато. Интереснее будет узнать, что вы можете сослаться на этот же самый объект через версию IComparer-интерфейса, сравнивающую строки, например:

IComparer<String> stringComparator = objectComparer;

На первый взгляд эта инструкция кажется нарушением всех правил обеспечения безопасности типов, которые только можно себе представить. Но если подумать о том, что именно делает интерфейс IComparer<T>, то этот подход обретает смысл. Цель метода Compare заключается в возвращении значения на основе сравнения переданных ему параметров. Если можно сравнить Objects, то, конечно же, можно будет сравнить и Strings, объекты которого являются всего лишь специализированными типами класса Objects. В конце концов, String должен справляться со всем, с чем справляется Object, — именно в этом и состоит цель наследования.

Но это по-прежнему отчасти похоже на предположение. Как компилятор C# узнает, что вы не собираетесь выполнять какую-либо операцию в отношении конкретного типа в программном коде, принадлежащем методу Compare, который может дать сбой, если вызвать метод через интерфейс, основанный на другом типе? Если еще раз посмотреть на определение интерфейса IComparer, можно увидеть, что перед параметром типа стоит квалификатор in:

public interface IComparer<in T>

{

int Compare(T x, T y);

}

Ключевое слово in сообщает компилятору C#, что вы можете передать методам в качестве параметра типа либо тип T, либо любой тип, являющийся производным от T. Вы не можете использовать T в качестве типа значения, возвращаемого из любых методов. По сути, это позволяет вам ссылаться на объект либо через интерфейс-обобщение, основанный на типе объекта, либо через интерфейс-обобщение, основанный на типе, являющемся производным от типа объекта. В общем виде, если тип A предоставляет некие операции, свойства или поля, то если тип B является производным от типа *A*, он также должен предоставлять те же самые операции (которые могут вести себя по-другому в случае их перегрузки), свойства и поля. Следовательно, должно быть абсолютно безопасно подставлять объект типа B вместо объекта типа *A*.

Ковариантность и контрвариантность могут показаться в мире обобщений некими дополнительными темами, но от них есть вполне определенная польза. Например, представляющий коллекцию класс-обобщение List<T> (в пространстве имен System.Collections.Generic) использует для реализации методов Sort и BinarySearch объекты IComparer<T>. Объект типа List<Object> может содержать коллекцию объектов любого типа, поэтому методам Sort и BinarySearch нужна возможность сортировать объекты любого типа. Без использования контравариантности методам Sort и BinarySearch потребовалось бы включать логику, определяющую реальные типы сортируемых или искомых элементов, а затем реализовывать механизм сортировки или поиска, соответствующий специфике типа. Но человеку, не связанному с математикой, будет нелегко вспомнить, что на самом деле представляют собой ковариантность и контравариантность. Способ, с помощью которого я это запомнил, основан на примерах, приведенных в данном разделе.

* Пример ковариантности. Если методы в интерфейсе-обобщении могут возвращать строки, то они могут возвращать и объекты. (Все строки являются объектами.)
* Пример контравариантности. Если методы в интерфейсе-обобщении могут получать в качестве параметров объекты, они могут получать и строковые параметры. (Если можно выполнить операцию путем использования объекта, значит, ту же самую операцию можно выполнить и с использованием строки, поскольку все строки являются объектами.)

ПРИМЕЧАНИЕ: Как и в случае с ковариантностью, контравариантными могут объявляться только интерфейсные и делегатные типы. Для классов-обобщений модификатор in не указывается.

ВАРИАНТЫ ЗАДАНИЙ

**Задание 1**

Разработать обобщенный класс, позволяющий работать с данными на примере стека/очереди. Продемонстрировать его работу.

**Задание 2**

Общее задание. Создать заданный в варианте обобщенный класс. Определить в классе индексаторы, конструкторы, деструктор, необходимые функции и заданные перегруженные операции. В методах класса для работы с аргументами используйте ref‐ и out‐параметры и операторы is или as, foreach. Используйте обработку исключительных ситуаций.

Индивидуальное задание (по вариантам)

|  |  |
| --- | --- |
| Вариант | Задание |
| 1 | Класс – одномерный массив. Дополнительно перегрузить следующие операции: \* – умножение массивов; int() – преобразование; ==, != – проверка на равенство, неравенство; <=, >= – сравнение |
| 2 | Класс – одномерный массив. Дополнительно перегрузить следующие операции: == – проверка на равенство; != – проверка на неравенство; ++ – инкремент массивов; ‐‐– декремент массивов |
| 3 | Класс – множество Сset. Дополнительно перегрузить следующие операции: + – добавить элемент в множество (типа set + item); \* – пересечение множеств, ==, != – проверка множеств на равенство, неравенство |
| 4 | Класс – очередь Сqueue. Дополнительно перегрузить следующие операции: + – добавить элемент; ‐ – извлечь элемент; bool () – проверка, пустая ли очередь; < – копирование одной очереди в другую с сортировкой в убывающем порядке; >– копирование одной очереди в другую с сортировкой в возрастающем порядке |
| 5 | Класс – множество Сset. Дополнительно перегрузить следующие операции: + – объединение множеств; <=, >= – сравнение множеств; int ()– мощность множества; &– пересечение |
| 6 | Класс – однонаправленный список list. Дополнительно перегрузить следующие операции: + – объединить два списка; ‐ – удалить элемент; ==, != – проверка на равенство, неравенство; ~– проверка, пустой ли список |
| 7 | Класс – матрица. Дополнительно перегрузить следующие операции: + – сложение матриц; () – доступ по индексу M(с, i); >, < – сравнение матриц; == – копирование матриц, != – проверка соответствия размеров матриц |
| 8 | Класс – время CVector. Дополнительно перегрузить следующие операции: + – добавление элемента; ++, ‐‐ – увеличение и уменьшения всех элементов вектора на один (СVector++); !=, == – проверка на неравенство, равенство |
| 9 | Класс – время CVector. Дополнительно перегрузить следующие операции: ‐ – разность двух векторов; + – сложение элементов вектора с целым; !=, == - проверка на неравенство, равенство |
| 10 | Класс – стек Сstack. Дополнительно перегрузить следующие операции: + – добавить элемент в стек; ‐ – извлечь элемент из стека; ~ – проверка, пустой ли стек; > и < – копирование одного стека в другой с сортировкой в возрастающем порядке |

ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ

1. Албахари, Дж. C#5.0. Справочник. Полное описание языка / Дж. Албахари, Б. Албахари. – 5-е изд. – М.: ООО «И.Д. Вильямс», 2014. – 1008 с.: ил.
2. Дейтел, П. Как программировать на Visual C# 2012 / П. Дейтел, Х. Дейтел. – 5-е изд. – СПб.: Питер, 2014. – 864 с.: ил.
3. Павловская Т.А. C#. Программирование на языке высокого уровня / Т.А. Павловская. – Учебник для вузов. – СПб.: Питер, 2014. – 432 с.: ил.
4. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен. – 6-е изд. – М.: ООО «И.Д. Вильямс», 2013. – 1312 с.: ил.
5. Шарп, Дж. Microsoft Visual C#. Подробное руководство / Дж. Шарп. – 8-е изд. – СПб.: Питер, 2017. – 848 с.: ил.
6. Шилдт, Г. C#4.0: Полное руководство / Г. Шилдт. – М.: ООО «И.Д. Вильямс», 2011. – 1056 с.: ил.

Преподаватель Е.В. Багласова

|  |
| --- |
| Рассмотрено на заседании цикловой  комиссии программного обеспечения информационных технологий №10  Протокол №\_\_\_\_\_от «\_\_\_»\_\_\_\_2017  Председатель ЦК *( )* С.В.Банцевич |