УО «МОГИЛЕВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ А.А. КУЛЕШОВА»

СОЦИАЛЬНО-ГУМАНИТАРНЫЙ КОЛЛЕДЖ

Дисциплина
«Конструирование программ и языки программирования»

«Реализация механизма создания коллекции и применение коллекций в программах»

Методические рекомендации к лабораторной работе № 17
(4 часа)

Могилев 2018

Составитель: Шлапаков А.В.

Понятие «Коллекция и типы коллекций». Методические указания по лабораторной работе №17 по дисциплине «Конструирование программ и языки программирования». Для учащихся очной формы обучения специальности 1–40 01 01 «Программное обеспечение информационных технологий.

Составитель: Шлапаков А.В.

Оглавление

Составитель: Шлапаков А.В.

1 Цель работы

научиться разрабатывать коллекции на языке программирования C# и применять их в программах.

2 Краткие теоретические сведения

2.1 Интерфейсы и типы коллекций

Большинство классов коллекций находятся в пространствах имен `System.Collections` и `System.Collections.Generic`. Классы обобщенных коллекций можно найти в пространстве имен `System.Collections.Generic`. Классы коллекций, специализированные для хранения определенного типа, находятся в пространстве имен `System.Collections.Specialized`. Классы коллекций, безопасных в отношении потоков, определены в пространстве имен `System.Collections.Concurrent`.

Конечно, существуют и другие способы разделения на группы классов коллекций. На основе интерфейсов, реализуемых классами коллекций, коллекции могут быть сгруппированы в списки, собственно коллекции и словари.

В табл. 2.1 описаны интерфейсы, реализуемые коллекциями и списками, а также их методы и свойства.

Таблица 2.1. Интерфейсы, их методы и свойства, важные для коллекций

Интерфейс	Описание
`IEnumerable<T>`	Интерфейс `IEnumerable<T>` необходим, когда с коллекцией используется оператор `foreach`. Этот интерфейс определяет метод `GetEnumerator()`, возвращающий перечислитель, который реализует `IEnumerator`.
`ICollection<T>`	`ICollection<T>` – это интерфейс, реализованный классами обобщенных коллекций. С его помощью можно получить количество элементов в коллекции (свойство `Count`) и скопировать коллекцию в массив (метод `CopyTo()`). Можно также добавлять и удалять элементы из коллекции (`Add()`, `Remove()`, `Clear()`).
`IList<T>`	Интерфейс `IList<T>` предназначен для создания списков, элементы которых доступны по своим позициям. Этот интерфейс определяет индексатор, а также способы вставки и удаления элементов в определенные позиции (методы Insert() и Remove()). `IList<T>` унаследован от `ICollection<T>`.
`ISet<T>`	Интерфейс `ISet<T>` появился в версии .NET 4. Этот интерфейс реализуется множествами. Он позволяет комбинировать различные множества в объединения, а также проверять, не пересекаются ли два множества. `ISet<T>` унаследован от `ICollection<T>`.
`IDictonary<TKey,TValue>`	Интерфейс `IDictonary<TKey,TValue>` реализуется обобщенными классами коллекций, элементы которых состоят из ключа и значения. С помощью этого интерфейса можно получать доступ ко всем ключам и значениям, извлекать элементы по индексатору типа ключа, также удалять и добавлять элементы.
`ILookup<TKey,TValue>`	Подобно поддерживает ключи и значения. Однако в этом случае коллекция может содержать множественные значения для одного ключа.
`IComparer<T>`	Интерфейс `IComparer<T>` реализован компаратором и

	используется для сортировки элементов внутри коллекции с помощью метода `Compare()`.
`IEqualityComparer<T>`	Интерфейс `IEqualityComparer<T>` реализован компаратором, который может быть применен к ключам словаря. Через этот интерфейс объекты могут быть проверены на предмет эквивалентности друг другу. В .NET 4 этот интерфейс также реализован массивами и кортежами.
`IProducerConsumerCollection<T>`	Интерфейс `IProducerConsumerCollection<T>` был добавлен в версию .NET 4 для поддержки новых, безопасных в отношении потоков классов коллекций.

2.2 Списки

Для динамических списков в .NET Framework предусмотрен обобщенный класс `List<T>`. Этот класс реализует интерфейсы `IList`, `ICollection`, `IEnumerable`, `IList<T>`, `ICollection<T>` и `IEnumerable<T>`.

В следующем примере члены класса `Racer` используются в качестве элементов, добавляемых к коллекции для представления гонщиков Формулы-1. В этом классе есть пять свойств: `Id`, `Firstname`, `Lastname`, `Country` и `Wins` (количество побед). Имя гонщика и количество его побед может быть передано конструктору для установки значений полей-членов. Метод `ToString()` переопределяется для возврата имени гонщика. Класс `Racer` реализует обобщенный интерфейс `IComparer<T>` для сортировки элементов – гонщиков, а также интерфейс `IFormattable`.

```
[Serializable]
public class Racer: IComparable<Racer>, IFormattable
{
   public int Id {get; private set;}
   public string FirstName {get; set;}
   public string LastName {get; set;}
   public string Country {get; set;}
   public int Wins {get; set;}
   public Racer(int id, string firstName, string lastName,
             string country = null, int wins = 0)
   {
      this.Id = id;
      this.FirstName = firstName;
      this.LastName = lastName;
      this.Country = country;
      this.Wins = wins;
   }
   public override string ToString()
   {
      return String.Format("{0} {1}", FirstName, LastName);
   }
   public string ToString(string format,
                          IFormatProvider formatProvider)
   {
      if (format == null) format = "N";
      switch (format.ToUpper())
      {
```

```
                case "N": // имя и фамилия
                   return ToString();
                case "F": // имя
                   return FirstName;
                case "L": // фамилия
                   return LastName;
                case "W": // количество побед
                   return String.Format("{0}, Побед: {1}",
                             ToString(), Wins);
                case "C": // страна
                   return    String.Format("{0},    Страна:    {1}",
ToString(),
                                          Country);
                case "A": // все вместе
                   return  String.Format  ("{0  },  {1}  Побед:  {2}",
ToString(),
                            Country, Wins);
             default:
                throw                new                FormatExcep-
tion(String.Format(formatProvider,
                  "Формат {0} не поддерживается", format));
          }
       }
       public string ToString(string format)
       {
          return ToString(format, null);
       }
       public int CompareTo(Racer other)
       {
          int compare = this.LastName.CompareTo(other.LastName);
          if (compare == 0)
             return this.FirstName.CompareTo(other.FirstName);
          return compare;
       }
    }
```

2.2.1 Создание списков

Создавать списочные объекты можно, вызывая конструктор по умолчанию. При объявлении обобщенного класса `List<T>` необходимо указывать тип хранимых значений. В приведенном ниже коде показано, как объявлять списки `List<T>` с элементами `int` и `Racer`. Класс `ArrayList` – это необобщенный список, принимающий элементы любого типа, производного от `Object`.

Конструктор по умолчанию создает пустой список. Как только элементы начинают добавляться в список, его емкость увеличивается до 4 элементов. При добавлении пятого элемента размер списка изменяется так, чтобы уместить 8 элементов. Если же и этого недостаточно, список вновь расширяется, на этот раз до 16 элементов. При каждом расширении емкость списка удваивается.

```
var intList = new List<int>();
var racers = new List<Racer>();
```

При изменении емкости списка вся коллекция целиком перемещается в новый блок

памяти. В реализации List<T> используется массив типа T**.** При перемещении создается новый массив, и с помощью Array.Copy() производится копирование элементов старого массива в новый. Чтобы сэкономить время, когда количество элементов, подлежащих размещению в списке, известно заранее, емкость можно определить в конструкторе. Ниже создается коллекция емкостью в 10 элементов. Если этой емкости будет недостаточно для размещения всех элементов, то она удваивается – сначала до 20, затем до 40 элементов.

```
List<int> intList = new List<int>(10);
```

С помощью свойства Capacity можно получать и устанавливать емкость коллекции:

```
intList.Capacity = 20;
```

Емкость коллекции – это не то же самое, что количество элементов в коллекции. Количество элементов в коллекции может быть прочитано в свойстве Count**.** Разумеется, емкость всегда больше или равна количеству элементов. До тех пор, пока ни один элемент не добавлен в коллекцию, количество равно 0.

```
Console.WriteLine(intList.Count);
```

Если вы завершили добавление элементов в коллекцию и не собираетесь добавлять новых, то можете избавиться от излишней емкости, вызвав метод TrimExcess(). Однако поскольку реорганизация коллекции требует времени, TrimExcess() не делает ничего, если количество элементов превышает 90% от текущей емкости.

```
intList.TrimExcess();
```

2.2.2 Инициализаторы коллекций

Присваивать значения коллекциям можно с помощью инициализаторов коллекций. Синтаксис инициализаторов коллекций подобен инициализаторам массивов. В инициализаторе коллекции значения присваиваются коллекции внутри фигурных скобок при инициализации коллекции:

```
var intList = new List < int > () {1, 2};
var stringList =
     new List <string>() {"one", "two"};
```

> *Инициализаторы коллекции не отражаются внутри кода IL скомпилированной сборки. Компилятор преобразует инициализатор коллекции в вызов метода Add() для каждого элемента списка инициализации.*

2.2.3 Добавление элементов

Добавлять элементы в список можно методом Add(), как показано ниже. Обобщенный параметрический тип определяет тип первого параметра метода Add()**.**

```
var intList = new List<int>();
intList.Add(1);'
intList.Add(2);
var stringList = new List<string>();
stringList.Add ("one");
stringList.Add("two");
```

Переменная racers определена как имеющая тип List<Racer>**.** С помощью операции new создается новый объект того же типа. Поскольку экземпляр класса List<Racer> создается с конкретным классом Racer**,** только объекты этого класса могут

быть добавлены методом Add(). В следующем примере кода создаются и добавляются в коллекцию пять гонщиков Формулы-1. Первые три добавляются с использованием инициализатора коллекции, а последние два – явным вызовом метода Add().

```
var graham = new Racer(7,"Graham","Hill","UK",14);
var emerson = new Racer(13,"Emerson","Fittipaldi",
"Brazil",14);
var mario = new Racer(16,"Mario","Andretti","USA",12);
var racers = new List <Racer>(20) {graham, emerson, mario};
racers.Add(new Racer(24,"Michael","Schumacher",
"Germany",91));
racers.Add(new Racer(27,"Mika","Hakkinen","Finland",20));
```

Применив метод AddRange() класса List<T>, можно добавить множество элементов в коллекцию за один прием. Метод AddRange() принимает объект типа IEnumerable<T>, так что допускается передавать массив, как показано ниже:

```
racers.AddRange(new Racer[] {
   new Racer("Niki","Lauda","Austria",25),
   new Racer("Alain","Prost","France",51)});
```

Инициализатор коллекции может использоваться только во время объявления коллекции. Метод AddRange() может быть вызван после инициализации коллекции.

Если при создании экземпляра списка известны все элементы, которые предполагается поместить в коллекцию, то конструктору класса можно передать любой объект, реализующий IEnumerable<T>. Это очень похоже на вызов метода AddRange():

```
var racers =
new List <Racer> (new Racer[] {
   new Racer (12,"Jochen","Rindt","Austria",6),
   new Racer(22,"Ayrton","Senna","Brazil",41) });
```

2.2.4 Вставка элементов

Для вставки элементов в определенную позицию коллекции служит метод **Insert** ():

```
racers.Insert(3, newRacer(6,"Phil","Hill","USA",3));
```

Метод InsertRange() предоставляет возможность вставки множества элементов, подобно тому, как это делает метод AddRange(), показанный ранее.

Если указывается индекс, превышающий количество элементов в коллекции, генерируется исключение типа ArgumentOutOfRangeException.

2.2.5 Доступ к элементам

Все классы, реализующие интерфейсы IList и IList<T>, предоставляют индексатор, так что к элементам можно обращаться с использованием индексатора, передавая ему номер элемента. Первый элемент доступен по индексу 0. Указывая racers[3], вы обратитесь к четвертому элементу списка:

```
Racer r1 = racers [3];
```

Получив количество элементов из свойства Count, вы можете выполнить цикл for для прохода по всем элементам коллекции, применяя индексатор для обращения к ним:

```
for (int i = 0; i < racers.Count; i++)
{
```

```
    Console.WriteLine(racers[i]);
}
```

Индексный доступ в классах коллекций возможен для `ArrayList`, `StringCollection` и `List<T>`.

Поскольку List<T> реализует интерфейс IEnumerable, проход по элементам коллекции можно также осуществлять с помощью оператора foreach:

```
foreach (Racer r in racers)
{
    Console.WriteLine(r);
}
```

В разделе 6.6 объясняется, как компилятор преобразует оператор `foreach` для использования интерфейсов `IEnumerable` и `IEnumerator`.

Вместо оператора `foreach` в классе `List<T>` также предусмотрен метод `ForEach()`, объявленный с параметром `Action<T>`:

```
public void ForEach(Action<T> action);
```

Реализация `ForEach()` показана ниже. `ForEach()` выполняет итерацию по каждому элементу коллекции и вызывает метод, переданный в качестве параметра, с каждым элементом.

```
public class List<T>: IList<T>
{
    private T[] items;
    //...
    public void ForEach(Action<T> action)
    {
        if (action == null) throw new ArgumentNullExcep-
tion("action");
        foreach (T item in items)
        {
            action(item);
        }
    }
    //...
}
```

Для того, чтобы `ForEach` передавать метод, `Action<T>` объявляется как делегат, определяющий метод, который возвращает `void` и принимает параметр `T`:

```
public delegate void Action<T>(T obj);
```

В случае списка элементов `Racer` обработчик для метода `ForEach()` должен быть объявлен с объектом `Racer` в качестве параметра и типом возврата `void`:

```
public void ActionHandler(Racer obj);
```

Поскольку одна из перегрузок метода `Console.WriteLine()` принимает в качестве параметра `Object`, можно передать адрес этого метода методу `ForEach()`, и каждый гонщик, содержащийся в коллекции, будет выведен на консоль:

```
racers.ForEach(Console.WriteLine);
```

Также можно написать лямбда-выражение, принимающее объект Racer в качестве параметра и выполняющее Console.WriteLine() в реализации. Ниже используется формат A с методом ToString() интерфейса IFormattable для отображения всей информации о гонщике:

```
racers.ForEach(r => Console.WriteLine ("{0 :A) ", r));
```

2.2.6 Удаление элементов

Элементы можно удалять по индексу либо передавая подлежащий удалению элемент. Ниже удаляется четвертый по порядку элемент:

```
racers.RemoveAt(3);
```

Чтобы удалить объект Racer, его можно также непосредственно передать методу Remove(). Удаление по индексу работает быстрее, поскольку в этом случае не приходится выполнять поиск удаляемого элемента по всей коллекции. Метод Remove() сначала ищет в коллекции индекс удаляемого элемента с помощью метода IndexOf(), а затем использует этот индекс для удаления элемента. IndexOf() сначала проверяет, реализует ли тип элемента интерфейс IEquatable<T>. Если это так, вызывается метод Equals() этого интерфейса для нахождения элемента в коллекции, совпадающего с переданным методу. Если же этот интерфейс не реализован, для сравнения элементов применяется метод Equals() класса Object. Реализация по умолчанию метода Equals() класса Object выполняет побитовое сравнение типов значений, но для ссылочных типов сравнивает только ссылки.

В приведенном ниже коде из коллекции удаляется гонщик, на которого ссылается переменная graham. Переменная graham была создана ранее, когда наполнялась коллекция. Поскольку интерфейс IEquatable<T> и метод Object.Equals() не переопределяются в классе Racer, нельзя создать новый объект с тем же содержимым, что и удаляемый элемент, и передать его методу Remove().

```
if (!racers.Remove(graham))
{
   Console.WriteLine("объект в коллекции не найден");
}
```

Метод RemoveRange() удаляет множество элементов из коллекции. Первый параметр специфицирует индекс, начиная с которого располагаются удаляемые элементы, а второй параметр задает количество удаляемых элементов.

```
int index = 3;
int count = 5;
racers.RemoveRange(index, count);
```

Чтобы удалить из коллекции все элементы с некоторыми специфическими характеристиками, вы можете использовать метод RemoveAll(). Этот метод применяет параметр Predicate<T>, о котором пойдет речь ниже, когда будет рассматриваться поиск элементов. Для удаления всех элементов из коллекции служит метод Clear(), определенный в интерфейсе ICollection<T>.

2.2.7 Поиск

Существуют различные способы поиска элементов в коллекции. Можно получить индекс найденного элемента или сам найденный элемент. Для использования доступны такие методы, как IndexOf(), LastIndexOf(), FindIndex(), FindLastIndex(), Find() и FindLast(). Для проверки существования элемента класс List<T> предлагает метод

Exists().

Метод IndexOf() в качестве параметра ожидает объект и возвращает индекс элемента, если таковой найден в коллекции. Если же элемент не найден, возвращается -1. Следует помнить, что IndexOf() для сравнения элементов использует интерфейс IEquatable<T>.

```
int index1 = racers.IndexOf(mario);
```

В методе **IndexOf**() можно также указать, что поиск не должен производиться по всей коллекции, а вместо этого задать индекс позиции, с которой следует начинать поиск, и количество элементов, которые необходимо просмотреть.

Вместо поиска определенного элемента с помощью метода IndexOf() можно искать элемент, обладающий определенными характеристиками, которые должны определяться в методе FindIndex(). Метод FindIndex() ожидает параметра типа Predicate:

```
public int FindIndex(Predicate<T> match);
```

Тип Predicate<T> – это делегат, который возвращает булевское значение и принимает тип T в качестве параметра. Этот делегат может быть использован аналогично делегату Action**,** показанному ранее, когда речь шла о методе **ForEach()**. Если предикат возвращает **true,** значит, обнаружено соответствие, и элемент найден. Если же возвращается **false,** значит, элемент не найден и поиск продолжается.

```
public delegate bool Predicate<T> (T obj);
```

С классом List<T>**,** использующим объект Racer вместо типа T**,** можно передавать адрес метода, возвращающего булевское значение и определяющего параметр типа Racer для метода FindIndex(). Для поиска первого гонщика из определенной страны можно создать класс FindCountry**,** как показано ниже. Метод FindCountryPredicate() имеет сигнатуру и тип возврата, определенный в делегате Predicate<T>**.** Метод Find() использует переменную country для поиска страны, которую можно передать конструктору класса,

```
public class FindCountry
{
   public FindCountry(string country)
   {
      this.country = country;
   }
   private string country;
   public bool FindCountryPredicate(Racer racer)
   {
      if (racer == 0) throw new ArgumentNullExcep-
tion("racer");
         return racer.Country == country;
   }
}
```

С помощью метода FindIndex() можно создать новый экземпляр класса FindCountry()**,** передать строку страны конструктору и передать адрес метода Find(). После успешного завершения метода FindIndex() переменная index2 содержит индекс первого элемента, где свойство Country гонщика установлено в Finland**:**

```
int index2 = racers.FindIndex(
```

```
    new FindCountry("Finland").FindCountryPredicate);
```

Вместо создания класса с методом-обработчиком можно также использовать лямбда-выражение. Результат будет в точности таким же, как и раньше. Теперь лямбда-выражение определяет реализацию поиска элемента, у которого свойство `Country` установлено

```
int index3 = racers.FindIndex(r => r.Country == "Finland");
```

Подобно `IndexOf()`, в методе `FindIndex()` можно также указать индекс позиции, с которой нужно начинать поиск, и количество элементов, которые следует просмотреть. Чтобы выполнить поиск индекса, начиная с последнего элемента коллекции, необходимо воспользоваться методом `FindLastIndex()`.

Метод `FindIndex()` возвращает индекс найденного элемента. Вместо получения индекса также можно получить непосредственно элемент коллекции. Метод `Find()` требует параметра типа `Predicate<T>`, как и метод `FindIndex()`. Ниже метод `Find()` выполняет поиск в списке первого гонщика, у которого свойство `Firstname` установлено в `Niki`. Разумеется, можно также применить метод `FindLast()` для поиска последнего элемента, удовлетворяющего предикату.

```
Racer r = racers.Find(r => r.FirstName == "Niki");
```

Для получения не одного, а всех элементов, удовлетворяющих критериям предиката, служит метод `FindAll()`. Метод `FindAll()` использует тот же делегат `Predicate<T>`, что и методы `Find()` и `FindIndex()`. Только метод `FindAll()` не останавливается при обнаружении первого найденного элемента, а вместо этого выполняет итерацию по всем элементам коллекции и возвращает все элементы, для которых предикат возвращает `true`.

Вызываемый здесь метод `FindAll()` возвращает всех гонщиков, у которых значение свойства `Wins` превышает `20`. Все гонщики, которые выигрывали более 20 гонок, попадают в список `bigWinners`:

```
List<Racer> bigWinners = racers.FindAll(r => r.Wins > 20);
```

Итерация по переменной `bigWinners` оператором `foreach` дает следующий результат:

```
foreach (Racer r in bigWinners)
{
   Console.WriteLine("{0:A}", r);
}
```

```
Michael Schumacher, Germany Побед: 91
Niki Lauda, Austria Побед: 25
Alain Prost, France Побед: 51
```

Результат не отсортирован, но это будет сделано позже.

2.2.8 Сортировка

Класс `List<T>` позволяет сортировать свои элементы с помощью метода `Sort()`, в котором реализован алгоритм быстрой сортировки.

Для использования доступно несколько перегрузок метода `Sort()`. Аргументы, которые могут ему передаваться – это делегат `Comparison<T>`, обобщенный интерфейс `IComparer<T>` и диапазон вместе с обобщенным интерфейсом `IComparer<T>`:

```
public void List<T>.Sort();
public void List<T>.Sort(Comparison<T>);
```

```
public void List<T>.Sort(IComparer<T>);
public void List<T>.Sort(Int32, Int32, IComparer<T>);
```

Использовать метод `Sort()` без аргументов можно только в том случае, когда элементы коллекции реализуют интерфейс `IComparable`.

Класс `Racer` реализует интерфейс `IComparable<T>` для сортировки гонщиков по фамилии:

```
racers.Sort();
racers.ForEach(Console.WriteLine);
```

Если сортировка должна быть выполнена иным способом, а не таким, который поддерживается по умолчанию типом элементов, потребуется воспользоваться другой техникой – например, передавать объект, реализующий интерфейс `IComparer<T>`.

Класс `RacerComparer` реализует интерфейс `IComparer<T>` для типа `Racer`. Этот класс позволяет сортировать по имени, фамилии, стране или количеству побед. Интересующий способ сортировки определяется внутренним перечислимым типом `CompareType`.

Тип `CompareType` устанавливается конструктором класса `RacerComparer`. Интерфейс `IComparer<Racer>` определяет метод `Compare()`, который необходим для сортировки. В реализации этого метода используется метод `CompareTo()` типов `string` и `int`.

```
public class RacerComparer: IComparer<Racer>
{
   public enum CompareType
   {
      Firstname,
      Lastname,
      Country,
      Wins
   }
   private CompareType compareType;
   public RacerComparer(CompareType CompareType)
   {
      this.CompareType = compareType;
   }
   public int Compare (Racer x, Racer y)
   {
      if(x == null) throw new ArgumentNullException("x");
      if(y == null) throw new ArgumentNullException("y") ;
      int result;
      switch (compareType)
      {
         case CompareType.Firstname:
            return x.Firstname.CompareTo(y.Firstname);
         case CompareType.Lastname:
            return x.Lastname.CompareTo(y.Lastname);
         case CompareType.Country:
            if((result = x.Country.CompareTo(y.Country) == 0)
               return x.Lastname.CompareTo(y.Lastname);
            else
               return result;
```

```
            case CompareType.Wins:
               return x.Wins.CompareTo(y.Wins);
            default:
               throw new ArgumentException(
        "Недопустимый тип для сравнения");
         }
      }
   }
```

Теперь экземпляр класса `RacerComparer` может быть использован вместе с методом `Sort()`. Передавая значение перечисления `RacerComparer.CompareType.Country`, мы сортируем коллекцию по значению свойства `Country`:

```
racers.Sort(new RacerComparer(RacerComparer.CompareType.Country));
racers.ForEach(Console.WriteLine);
```

Другой способ сортировки состоит в применении перегруженного метода `Sort()`, который принимает делегат `Comparison<T>`:

```
public void List<T>.Sort(Comparison<T>);
```

`Comparison<T>` представляет собой делегат метода, принимающего два параметра типа `T` и возвращающего тип `int`. Если значения параметров эквиваленты, метод должен вернуть 0. Если первый параметр меньше второго, должно быть возвращено значение меньше нуля; в противном случае возвращается значение больше нуля.

```
public delegate int Comparison<T>(T x, T y);
```

Теперь методу `Sort()` можно передать лямбда-выражение, чтобы выполнить сортировку по количеству побед. Два параметра имеют тип `Racer`, и в этой реализации свойства Wins сравниваются с помощью метода `CompareTo()`. В реализации `r2` и `r1` используются в обратном порядке, так что количество побед сортируется по убыванию. После вызова метода полный список гонщиков сортируется по имени гонщика.

```
racers.Sort((r1, r2) = > r2.Wins.CompareTo(rl.Wins));
```

С помощью метода `Reverse()` можно поменять порядок элементов коллекции на противоположный.

2.2.9 Преобразование типов

С помощью метода `ConvertAll<TOutput>()` из `List<T>` все типы коллекции могут быть преобразованы в другой тип. Метод `ConvertAll<TOutput>()` использует делегат `Converter`, определенный следующим образом:

```
public sealed delegate TOutput Converter<TInput,T0utput>
          (TInput from);
```

В этом преобразовании используются обобщенные типы `TInput` и `TOutput`. Тип `TInput` – это аргумент делегата метода, а `TOutput` – его тип возврата.

В рассматриваемом примере все элементы `Racer` должны быть преобразованы в тип `Person`. В то время как тип `Racer` содержит имя, фамилию, страну и количество побед, тип Person включает лишь полное имя. При таком преобразовании страна гонщика и количество побед могут быть проигнорированы, но полное имя должно быть преобразовано.

```
[Serializable]
```

```
public class Person
{
   private string name;
   public Person(string name)
   {
      this.name = name;
   }
   public override string ToString()
   {
      return name;
   }
}
```

Собственно преобразование выполняется вызовом метода racers.ConvertAll.Person>(). Аргумент этого метода определен как лямбда-выражение с аргументом типа Racer и типом возврата Person**.** В реализации лямбда-выражения создается и возвращается новый объект Person**.** Для этого его конструктору передаются FirstName и LastName**.**

```
List<Person> persons =
   racers.ConvertAllPerson> (r => new Person(r.FirstName + "
" + r.LastName));
```

Результатом преобразования будет список, содержащий преобразованные объекты Person**,** а точнее – persons типа List<Person>**.**

2.2.10 Коллекции, доступные только для чтения

После того, как коллекции созданы, они доступны для чтения и записи. Конечно, они должны быть таковыми, иначе вы не сможете наполнить их значениями. Тем не менее, после заполнения коллекции имеется возможность создать коллекцию, доступную только для чтения. Коллекция List<T> имеет метод AsReadOnly()**,** возвращающий объект типа ReadOnlyCollection<T>**.** Класс ReadOnlyCollection<T> реализует те же интерфейсы, что и **List<T>,** но все методы и свойства, которые изменяют коллекцию, генерируют исключение NotSupportedException**.**

2.3 Очередь

Очередь (queue) – это коллекция, в которой элементы обрабатываются по схеме “первый вошел, первый вышел” (first in, first out – FIFO). Элемент, вставленный в очередь первым, первым же и читается. Примерами очередей могут служить очередь в аэропорту, очередь претендентов на трудоустройство, очередь печати принтера либо циклическая очередь потоков на выделение ресурсов процессора. Часто встречаются очереди, в которых элементы обрабатываются по-разному, в соответствии с приоритетом. Например, в очереди в аэропорту пассажиры бизнес-класса обслуживаются перед пассажирами эконом-класса. Здесь может использоваться несколько очередей – по одной для каждого приоритета. В аэропорту это можно видеть наглядно, поскольку там предусмотрены две стойки регистрации для пассажиров бизнес-класса и эконом-класса. То же справедливо и для очередей печати и диспетчера потоков. У вас может быть массив списка очередей, где элемент массива означает приоритет. Внутри каждого элемента массива будет очередь, и обработка будет выполняться по принципу FIFO.

Очередь реализуется с помощью класса Queue<T> из пространства имен System.Collections.Generic**.** Внутри класс Queue<T> использует массив типа T**,** подобно тому, как это делает класс List<T>**.** Он реализует интерфейсы IEnumerable<T> и

ICollection**,** но не ICollection<T>**.** Интерфейс ICollection<T> не реализован, поскольку он определяет методы Add() и Remove()**,** которые не должны быть доступны для очереди.

Класс Queue<T> не реализует интерфейс IList<T>, поэтому обращаться к элементам очереди через индексатор нельзя. Очередь позволяет лишь добавлять элементы, при этом элемент помещается в конец очереди (методом Enqueue()), а таже получать элементы из головы очереди (методом Dequeue()).

На рис. 2.1 показаны элементы очереди. Метод Enqueue() добавляет элементы в конец очереди; элементы читаются и удаляются на другом конце очереди с помощью метода Dequeue(). Каждый последующий вызов метода Dequeue() удаляет следующий элемент очереди.

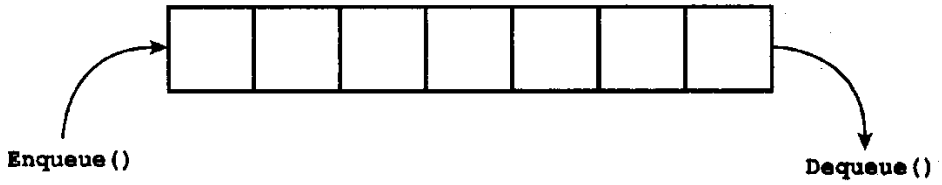

Рисунок 2.1 – Простая очередь

Методы класса Queue<T> описаны в табл. 2.2.

Таблица 2.2. Члены класса Queue<T>

Избранные члены класса Queue<T>	**Описание**
Count	Свойство Count возвращает количество элементов в очереди.
Enqueue()	Метод Enqueue() добавляет элемент в конец очереди.
Dequeue()	Метод Dequeue() читает и удаляет элемент из головы очереди. Если на момент вызова метода Dequeue() элементов в очереди больше нет, генерируется исключение InvalidOperationException.
Peek()	Метод Peek() читает элемент из головы очереди, но не удаляет его.
TrimExcess()	Метод TrimExcess() изменяет емкость очереди. Метод удаляет элементы из очереди, но не изменяет ее емкости. TrimExcess() позволяет избавиться от пустых элементов в начале очереди.

При создании очередей можно использовать конструкторы, подобные тем, что применялись с типом List<T>**.** Конструктор по умолчанию создает пустую очередь, но конструктор можно также использовать для указания начальной емкости. По мере добавления элементов в очередь емкость растет, позволяя разместить сначала 4, затем 6, 16 и 32 элемента, если емкость не определена. Подобно классу List<T>**,** емкость очереди при необходимости удваивается. Конструктор по умолчанию необобщенного класса Queue отличается тем, что создает начальный массив из 32 пустых элементов. Используя перегруженные конструкторы, можно передавать любую коллекцию, реализующую интерфейс IEnumerable<T>**,** содержимое которой копируется в очередь.

Примером приложения, демонстрирующего использование класса `Queue<T>`**,** может быть программа управления документами. Один поток используется для добавления элементов в очередь, а другой читает документы из нее и обрабатывает их.

```
public class Document
{
   public string Title {get; private set;}
   public string Content (get; private set;}
   public Document (string title, string content)
   {
      this.Title = title;
      this.Content = content;
   }
}
```

Класс `DocumentManager` – тонкая оболочка вокруг класса `Queue<T>`**.** Класс `DocumentManager` определяет то, как обрабатываются документы: добавление документов в очередь методом `AddDocument()` и получение их из очереди методом `GetDocument()`**.**

Внутри метода `AddDocument()` документ добавляется в конец очереди с помощью метода `Enqueue()`. Внутри `GetDocument()` методом `Dequeue()` читается первый документ в очереди. Поскольку несколько потоков могут обращаться к `DocumentManager` параллельно, доступ к очереди блокируется с помощью оператора `lock`**.**

`IsDocumentAvailable` – доступное только для чтения булевское свойство, которое возвращает `true`**,** если в очереди есть документы, и `false` – если нет.

```
public class DocumentManager
{
   private readonly Queue<Document> documentQueue =
       new Queue<Document>();
   public void AddDocument(Document doc)
   {
      lock (this)
      {
         documentQueue.Enqueue(doc);
      }
   }
   public Document GetDocument ()
   {
      Document doc = null;
      lock (this)
      {
         doc = documentQueue.Dequeue();
      }
      return doc;
   }
   public bool IsDocumentAvailable
   {
     get
     {
        return documentQueue.Count > 0;
     }
```

```
    }
}
```

Класс `ProcessDocuments` обрабатывает документы из очереди в отдельном потоке. Единственный его метод, доступный извне – это `Start()`. В этом методе инициируется новый поток. Для запуска потока создается объект `ProcessDocuments`, а метод `Run()` определен как стартовый метод этого потока. `ThreadStart` – это делегат, который ссылается на метод, запускаемый потоком. После создания объекта `Thread` поток запускается вызовом метода `Thread.Start()`.

В методе `Run()` класса `ProcessDocuments` определен бесконечный цикл. Внутри этого цикла с использованием свойства `IsDocumentAvailable` выясняется, есть ли в очереди какой-нибудь документ. Если в очереди имеется документ, он выбирается из `DocumentManager` и обрабатывается. В данном случае обработка состоит просто в выводе его на консоль. В реальном приложении документ может быть записан в файл, сохранен в базе данных либо передан по сети.

```
public class ProcessDocuments
{
   public static void Start(DocumentManager dm)
   {
      new Thread(new ProcessDocuments(dm).Run).Start();
   }
   protected ProcessDocuments(DocumentManager dm)
   {
      documentManager = dm;
   }
   private DocumentManager documentManager;
   protected void Run()
   {
      while (true)
      {
         if (documentManager.IsDocumentAvailable)
         {
            Document doc = documentManager.GetDocument();
            Console.WriteLine("Обработка              документа
{0}",doc.Title);
         }
         Thread.Sleep (new Random().Next(20));
      }
   }
}
```

В методе `Main()` приложения создается экземпляр объекта `DocumentManager` и запускается поток обработки документов. Затем создаются 1000 документов, которые добавляются к `DocumentManager`.

```
class Program
{
   static void Main()
   {
      var dm = new DocumentManager();
      ProcessDocuments.Start(dm);
```

```
            // Создать документы и добавить их в DocumentManager
            for (int i = 0; i < 1000; i++)
            {
                Document doc = new Document("Doc
"+i.ToString(),"content");
                dm.AddDocument(doc);
                Console.WriteLine("Добавлен документ
{0}",doc.Title);
                Thread.Sleep(new Random().Next(20));
            }
        }
    }
```

После запуска этого приложения документы будут добавляться и удаляться из очереди, и вы получите вывод вроде показанного ниже:

```
Добавлен документ Doc 279
Обработка документа Doc 236
Добавлен документ Doc 280
Обработка документа Doc 237
Добавлен документ Doc 281
Обработка документа Doc 238
Обработка документа Doc 239
Обработка документа Doc 240
Обработка документа Doc 241
Добавлен документ Doc 282
Обработка документа Doc 242
Добавлен документ Doc 283
Обработка документа Doc 243
```

Реальный сценарий, подобный описанному выше, может предусматривать создание приложения для обработки документов, полученных через веб-службу.

2.4 Стек

Стек (stack) – это еще один контейнер, очень похожий на очередь. Для доступа к элементам в нем используются другие методы.

Элемент, добавленный к стеку последним, читается первым. Стек – это контейнер, работающий по принципу “последний вошел, первый вышел” (last in, first out – LIFO).

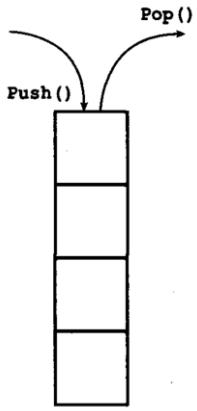

Рисунок 2.2 – Простой стек

На рис. 2.2 показано представление стека, где метод `Push()` добавляет элемент, а метод `Pop()` – получает элемент, добавленный последним.

Подобно классу `Queue<T>`, класс `Stack<T>` реализует интерфейсы `IEnumerable<T>` и `ICollection`.

Члены класса `Stack<T>` перечислены в табл. 2.3

Таблица 2.3. Члены класса `Stack<T>`

Избранные члены класса `Stack<T>`	**Описание**

`Count`	Свойство `Count` возвращает количество элементов в стеке.
`Push()`	Метод `Push()` добавляет элемент в вершину стека.
`Pop()`	Метод `Pop()` удаляет и возвращает элемент из вершины стека. Если стек пуст, генерируется исключение `InvalidOperationException`.
`Peek()`	Метод `Peek()` возвращает элемент из вершины стека, но не удаляет его.
`Contains()`	Метод `Contains()` проверяет наличие элемента в стеке и возвращает `true` в случае нахождения его там.

В следующем примере с помощью метода **Push**() в стек помещаются три элемента. Оператором **foreach** выполняется итерация по всем элементам с использованием интерфейса **IEnumerable.** Перечислитель стека не удаляет элементов из него – он только возвращает их.

```
var alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push ('B');
alphabet.Push('C');
foreach (string item in alphabet)
{
   Console.Write(item) ;
}
Console.WriteLine ();
```

Поскольку элементы читаются в порядке от последнего добавленного к первому, получаем следующий результат:

```
CBA
```

Чтение элементов с помощью перечислителя не изменяет состояния элементов. С помощью метода `Pop()` каждый прочитанный элемент также удаляется из стека. Способ итерации по коллекции циклом `while` заключается в проверке количества оставшихся элементов через свойство `Count`.

```
var alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');
Console.Write("Первая итерация: ");
foreach (string item in alphabet)
{
   Console.Write(item);
}
Console.WriteLine();
Console.Write("Вторая итерация: ");
while (alphabet.Count > 0)
{
   Console.Write(alphabet.Pop());
}
Console.WriteLine();
```

В результате два раза получаем `CBA` – по одному для каждой итерации. После второй итерации стек оказывается пуст, поскольку использовался метод `Pop()`.

```
Первая итерация: CBA
Вторая итерация: CBA
```

2.5 Связный список

Класс `LinkedList<T>` представляет собой двухсвязный список, в котором каждый элемент ссылается на следующий и предыдущий, как показано на рис. 2.3.

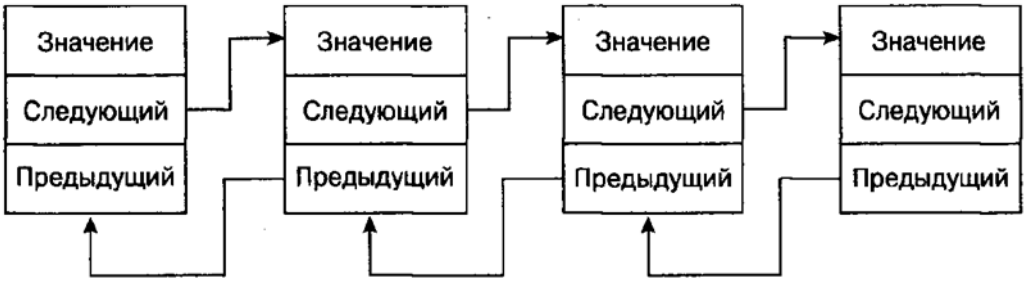

Рисунок 2.3 – Простой двухсвязный список

Преимущество связного списка проявляется в том, что операция вставки элемента в середину выполняется очень быстро. При этом только ссылки `Next` (следующий) предыдущего элемента и `Previous` (предыдущий) следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент. В классе `List<T>` при вставке нового элемента все последующие должны быть сдвинуты.

Естественно, у связных списков есть и свои недостатки. Так, например, все элементы связных списков доступны лишь друг за другом. Поэтому для нахождения элемента, находящегося в середине или конце списка, требуется довольно много времени.

Связный список не может просто хранить элементы внутри себя. Вместе с каждым из них ему необходимо иметь информацию о следующем и предыдущем элементах. Вот почему `LinkedList<T>` содержит элементы типа `LinkedListNode<T>`**.** С помощью класса `LinkedListNode<T>` появляется возможность обратиться к предыдущему и последующему элементам списка. Класс `LinkedListNode<T>` определяет свойства `List`**,** `Next`**,** `Previous` и `Value`**.** Свойство `List` возвращает объект `LinkedList<T>`**,** ассоциированный с узлом. Свойства `Next` и `Previous` предназначены для итераций по списку и для доступа к следующему и предыдущему элементам. Свойство `Value` типа `T` возвращает элемент, ассоциированный с узлом.

Сам класс `LinkedList<T>` определяет члены для доступа к первому **(**`First`**)** и последнему **(**`Last`**)** элементам в списке, для вставки элементов в определенные позиции (`AddAfter()`**,** `AddBefore()`**,** `AddFirst()`**,** `AddLast()`), для удаления элементов из заданных позиций (`Remove()`, `RemoveFirst()`, `RemoveLast()`) и для нахождения элементов, начиная поиск либо с начала (`Find()`), либо с конца (`FindLast()`) списка.

В примере приложения демонстрации связных списков используется связный список вместе с простым списком. Связный список содержит документы, как и в примере с очередью, но на этот раз документы имеют ассоциированный с ними дополнительный приоритет. Документы сортируются внутри связного списка в соответствии со своими приоритетами. Если множество документов имеют один и тот же приоритет, то такие элементы сортируются по времени их вставки.

На рис. 2.4 показаны коллекции из рассматриваемого примера приложения. `LinkedList<Document>` – это связный список, содержащий все объекты типа `Document`**.**

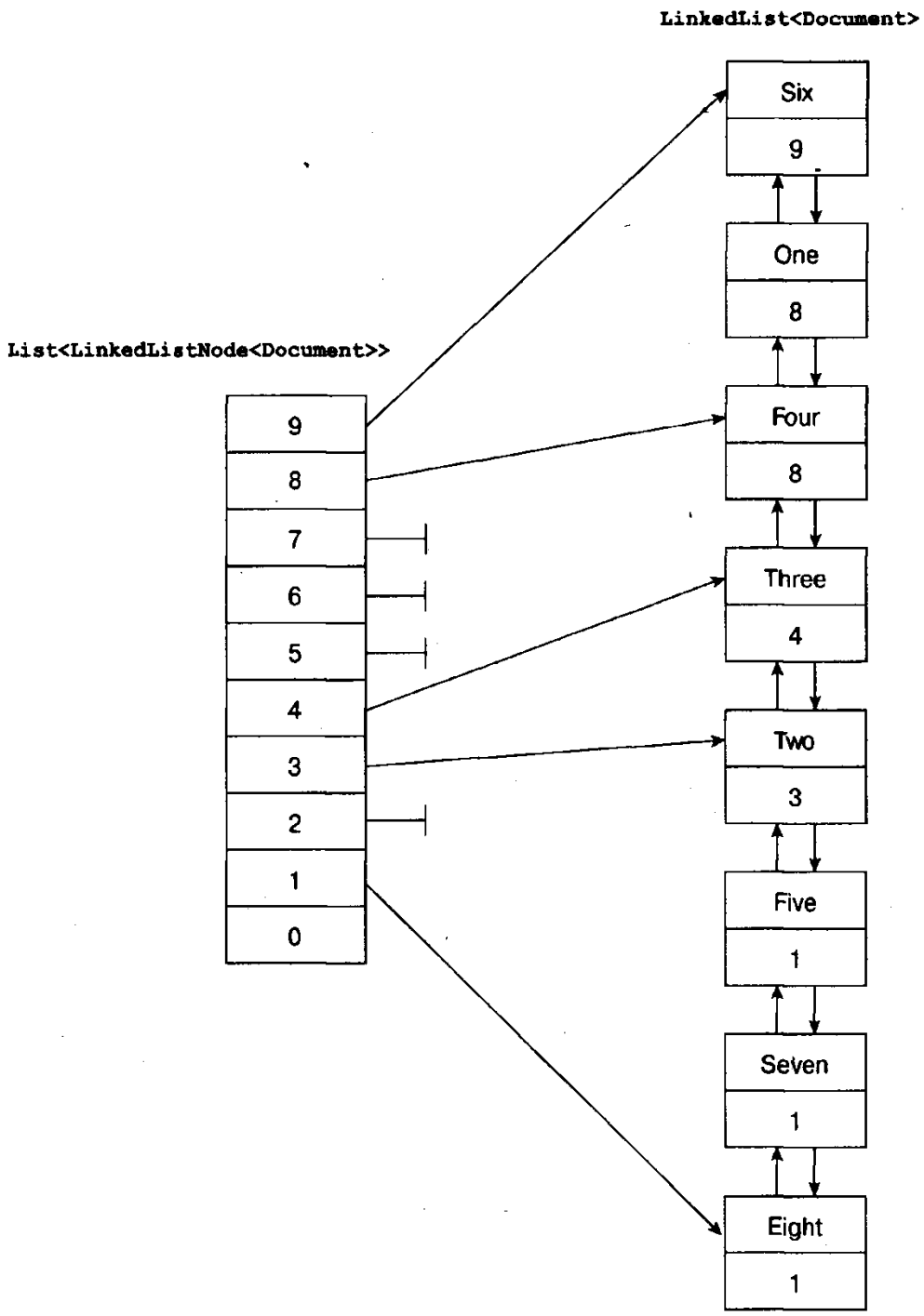

Рисунок 2.4 – Коллекции в классе PriorityDocumentManager

На рисунке представлены заголовки и приоритеты документов. Заголовок указывает, когда документ был добавлен в список. Первый добавленный документ имеет заголовок `"One"`, второй – `"Two"` и т.д. Как видите, документы `One` и `Four` имеют одинаковый приоритет – 8, но поскольку `One` был добавлен перед `Four`, он располагается в списке раньше. Когда новые документы добавляются в связный список, они должны размещаться после последнего документа с таким же приоритетом. Коллекция `LinkedList<Document>` содержит элементы типа `LinkedListNode<Document>`.

Класс `LinkedListNode<T>` добавляет свойства `Next` и `Previous`, позволяющие перемещаться от одного узла к следующему. Для обращения к этим элементам `List<T>`

определен как `List<LinkedListNode<Document>>`. Для быстрого доступа к последнему документу с любым приоритетом коллекция `List<LinkedListNode>` содержит до 10 элементов, каждый из которых ссылается на последний документ с каждым из приоритетов. При последующем обсуждении ссылка на последний документ каждого приоритета будет называться *узлом приоритета.*

Расширим класс `Document` из предыдущего примера дополнительным полем приоритета. Приоритет будет устанавливаться в конструкторе класса.

```
public class Document
{
   public string Title {get; private set;}
   public string Content {get; private set;}
   public byte Priority {get; private set;}
   public Document(string title, string content, byte priori-
ty = 0)
   {
      this.Title = title;
      this.Content = content;
      this.Priority = priority;
   }
}
```

Центральной частью этого решения будет класс `PriorityDocumentManager`. Этот класс очень легко использовать. С помощью общедоступного интерфейса этого класса в связный список могут добавляться новые элементы `Document`, а также извлекаться первый документ. В целях тестирования также предусмотрен метод для отображения всех элементов коллекции, как они связаны в списке.

Класс `PriorityDocumentManager` содержит две коллекции. Коллекция типа `LinkedList<Document>` содержит все документы.

Коллекция типа `List<LinkedListNode<Document>>` содержит ссылки на максимум 10 элементов, которые являются точками входа для добавления новых документов ^определенным приоритетом. Обе переменных коллекций инициализируются в конструкторе класса `PriorityDocumentManager`. Список коллекции также инициализируется `null`.

```
public class PriorityDocumentManager
{
private readonly LinkedList<Document> documentList;
// приоритеты от 0 до 9
private   readonly   List<LinkedListNode<Document>>   priori-
tyNodes;
public PriorityDocumentManager()
{
   documentList = new LinkedList<Document>();
   priorityNodes = new List<LinkedListNode<Document>>(10);
   for (int i = 0; i < 10; i++)
   {
      priorityNodes.Add(new LinkedListNode<Document>(null));
   }
}
```

Частью общедоступного интерфейса класса является метод `AddDocument()`. Он не делает ничего кроме вызова приватного метода `AddDocumentToPriorityNode()`. При-

чина помещения этой реализации внутрь другого метода заключается в том, что AddDocumentToPriorityNode() может вызываться рекурсивно, в чем вы вскоре убедитесь.

```
public void AddDocument(Document d)
{
   if (d == null) throw new ArgumentNullException("d");
   AddDocumentToPriorityNode(d, d.Priority);
}
```

Первое действие, выполняемое в реализации AddDocumentToPriorityNode() – это проверка нахождения указанного приоритета в пределах допустимых значений. В данном случае эти пределы простираются от 0 до 9. Если передается неверное значение, генерируется исключение ArgumentException.

Затем производится проверка, существует ли узел приоритета с тем же значением приоритета, что было передано. Если такого узла в коллекции нет, рекурсивно вызывается AddDocumentToPriorityNode() с уменьшенным значением приоритета, чтобы найти узел приоритета с ближайшим меньшим значением.

Если узел приоритета с тем же или меньшим значением приоритета не найден, значит, документ может быть добавлен в конец связного списка вызовом метода AddLast(). Кроме того, на этот узел связного списка заводится ссылка узла приоритета, отвечающего за приоритет документа.

Если соответствующий узел приоритета существует, можно получить положение внутри связного списка, куда должен быть вставлен новый документ. Здесь следует отличать случай, когда узел приоритета с нужным значением приоритета уже существует, от случая, когда имеется узел приоритета, ссылающийся на документ с меньшим приоритетом. Поскольку узел приоритета всегда должен ссылаться на последний документ с определенным приоритетом, ссылка узла приоритета уже должна быть установлена. Если же существует только узел приоритета, ссылающийся на документ с меньшим приоритетом, все несколько усложняется. В этом случае документ должен быть вставлен перед всеми документами с тем же приоритетом, который имеет узел приоритета. Чтобы получить первый документ того же приоритета, в цикле while осуществляется проход по всем узлам связного списка с использованием свойства Previous до тех пор, пока не будет достигнут узел связного списка, имеющий другой приоритет. Подобным образом получается позиция, куда должен быть вставлен документ и установлен узел приоритета.

```
private void AddDocumentToPriorityNode(Document doc, int pri-
ority)
{
   if (priority >9|| priority < 0)
      throw new ArgumentException(
         "Приоритет должен находиться в пределах от 0 до 9");
   if (priorityNodes[priority].Value == null)
   {
      --priority;
      if(priority >= 0)
      {
         // проверить следующий меньший приоритет
         AddDocumentToPriorityNode(doc, priority);
      }
      else//теперь нет узлов приоритетов с тем же или меньшим
           //приоритетом  добавить документ в конец
      {
```

```
            documentList.AddLast(doc);
            priorityNodes[doc.Priority] = documentList.Last;
         }
         return;
      }
      else // узел приоритета существует
      {
         LinkedListNode<Document>     prioNode     =     priori-
tyNodes[priority];
         if (priority == doc.Priority)
//узел приоритета с тем же значением приоритета уже существует
         {
            documentList.AddAfter(prioNode, doc);
//установить узел приоритета в последний документ с тем же
            //приоритетом
            priorityNodes[doc.Priority] = prioNode.Next;
         }
         else // существует только узел приоритета с меньшим
              // значением приоритета
         {
            // получить первый узел с меньшим приоритетом
            LinkedListNode<Document> firstPrioNode = prioNode;
            while (firstPrioNode.Previous != null &&
               firstPrioNode.Previous.Value.Priority ==
               prioNode.Value.Priority)
            {
               firstPrioNode = prioNode.Previous;
               prioNode = firstPrioNode;
            }
            documentList.AddBefore(firstPrioNode, doc);
            // установить узел приоритета в новое значение
            priorityNodes[doc.Priority]       =       firstPrioN-
ode.Previous;
         }
      }
   }
```

После этого остается обсудить лишь простые методы. `DisplayAllNodes()` просто выполняет цикл `foreach` для вывода на консоль приоритета и заголовка каждого документа.

Метод `GetDocument()` возвращает первый документ (с максимальным приоритетом) из связного списка и удаляет его оттуда.

```
   public void DisplayAllNodes()
   {
      foreach (Document doc in documentList)
      {
         Console.WriteLine("приоритет: (0), заголовок {1}",
             doc.Priority, doc.Title);
      }
   }
   // возвращает документ с максимальным приоритетом
```

```
// (первый в связном списке)
public Document GetDocument()
{
   Document doc = documentList.First.Value;
   documentList.RemoveFirst ();
   return doc;
}
```

В методе `Main()` класс `PriorityDocumentManager` используется для демонстрации своей функциональности. В связный список добавляются восемь новых документов с различными приоритетами, после чего заполненный список отображается.

```
static void Main()
{
   PriorityDocumentManager pdm = new PriorityDocumentManag-
er();
   pdm.AddDocument(new Document("one", "Sample", 8));
   pdm.AddDocument(new Document("two", "Sample", 3) );
   pdm.AddDocument(new Document("three", "Sample", 4));
   pdm.AddDocument(new Document("four", "Sample", 8));
   pdm.AddDocument(new Document("five", "Sample", 1));
   pdm.AddDocument(new Document("six", "Sample", 9) );
   pdm.AddDocument(new Document("seven", "Sample", 1));
   pdm.AddDocument(new Document("eight", "Sample", 1));
   pdm.DisplayAllNodes();
}
```

Просматривая результат работы этой программы, легко убедиться, что документы в списке отсортированы сначала по приоритетам, а затем — по времени добавления:

```
приоритет:9, заголовок six
приоритет:8, заголовок one
приоритет:8, заголовок four
приоритет:4, заголовок three
приоритет:3, заголовок two
приоритет:1, заголовок five
приоритет:1, заголовок seven
приоритет:1, заголовок eight
```

2.6 Сортированный список

Если нужна коллекция, отсортированная по ключу, можно воспользоваться `SortedList<TKey, TValue>`. Этот класс сортирует элементы на основе значения ключа. Можно использовать не только любой тип значения, но также и любой тип ключа.

В приведенном ниже примере создается сортированный список, в котором как ключ, так и значение имеют тип `string`. Конструктор по умолчанию создает пустой список, в который с помощью метода `Add()` добавляются две книги. Применяя перегруженные конструкторы, можно указать емкость списка, а также передать объект, который реализует интерфейс `IComparer<TKey>`, используемый для сортировки элементов в списке.

Первый параметр метода `Add()` – ключ (заголовок книги); второй параметр – значение (номер ISBN). Для добавления элементов в список вместо метода `Add()` можно применить индексатор. Индексатор требует ключ в качестве параметра индекса. Если такой ключ уже существует, метод `Add()` генерирует исключение `ArgumentException`. Если

же то же значение ключа применяется с индексатором, то новое значение заменяет старое.

> *SortedList<TKey, TValue> допускает только одно значение на ключ. Если нужно иметь несколько значений на ключ, следует использовать Lookup<TKey, TElement>.*

```
var books = new SortedList<string, string>();
books.Add(C# 2008 Wrox Box", "978-0-470-047205-7");
books.Add("Professional ASP.NET MVC 1.0", "978-0-470-38461-
9");
books["Beginning Visual C# 2008"] = "978-0-470-19135-4";
books["Professional C# 2008"] = "978-0-470-19137-6";
```

С помощью оператора `foreach` можно выполнить итерацию по списку. Элементы, возвращенные перечислителем, имеют тип `KeyValuePair<TKey, TValue>`**,** который содержит как ключ, так и значение. Ключ доступен через свойство `Key`**,** а значение – через свойство `Value`**.**

```
foreach (KeyValuePair<string, string> book in books)
{
   Console.WriteLine("{0}, {1}", book.Key, book.Value);
} '
```

Эта итерация отображает заголовки книг и номера ISBN, упорядоченные по ключу:

```
Beginning Visual C# 2008, 978-0-470-19135-4
C# 2008 Wrox Box, 978-0-470-047205-7
Professional ASP.NET MVC 1.0, 978-0-470-38461-9
Professional C# 2008, 978-0-470-19137-6
```

Свойства `Keys` и `Values` позволяют обращаться сразу ко всем ключам и значениям. Свойство `Values` возвращает `IList<TValue>`, а свойство `Keys` – `IList<TKey>`, так что эти свойства можно использовать вместе с `foreach`:

```
foreach (string isbn in books.Values)
{
   Console.WriteLine(isbn);
}
foreach (string title in books.Keys)
{
   Console.WriteLine(title);
}
```

Первый цикл отобразит значения, второй – ключи:

```
978-0-470-19135-4
978-0-470-047205-7
978-0-470-38461-9
978-0-470-19137-6
Beginning Visual C# 2008
C# 2008 Wrox Box P
rofessional ASP.NET MVC 1.0
Professional C# 2008
```

Если вы попытаетесь обратиться к элементу по индексатору, передав несуществующий ключ, будет сгенерировано исключение типа `KeyNotFoundException`**.** Чтобы из-

бежать этого исключения, можно воспользоваться методом `ContainsKey()`, который возвращает `true`, если переданный ключ существует в коллекции, или же вызвать метод `TryGetValue()`, который пытается получить значение, но не генерирует исключение, если значение отсутствует:

```
string isbn;
string title = "Professional C# 7.0";
if (!books.TryGetValue(title, out isbn))
{
   Console.WriteLine("Книга {0} не найдена", title);
}
```

2.7 Словари

Словарь (dictionary) представляет собой сложную структуру данных, позволяющую обеспечить доступ к элементам по ключу. Главное свойство словарей – быстрый поиск на основе ключей. Можно также свободно добавлять и удалять элементы, подобно тому,-как это делается в `List<T>`, но без накладных расходов производительности, связанных с необходимостью смещения последующих элементов в памяти.

На рис. 2.5 представлена упрощенная модель словаря. Здесь ключами словаря служат идентификаторы сотрудников, такие как B4711. Ключ трансформируется в хеш. В хеше создается число для ассоциации индекса со значением. После этого индекс содержит ссылку на значение. Изображенная модель является упрощенной, поскольку существует возможность того, что единственное вхождение индекса может быть ассоциировано с несколькими значениями, и индекс может храниться в виде дерева.

В .NET Framework предлагается несколько классов словарей. Главный класс, который можно использовать – это `Dictionary<TKey, TValue>`.

2.7.1Тип ключа

Тип, используемый в качестве ключа словаря, должен переопределять метод `GetHashCode()` класса `Object`. Всякий раз, когда класс словаря должен найти местоположение элемента, он вызывает метод `GetHashCode()`.

Целое число, возвращаемое этим методом, используется словарем для вычисления индекса, куда помещен элемент. Мы не станем углубляться в подробности работы этого алгоритма. Единственное, что следует знать – это то, что он использует простые числа, так что емкость словаря всегда выражается простым числом.

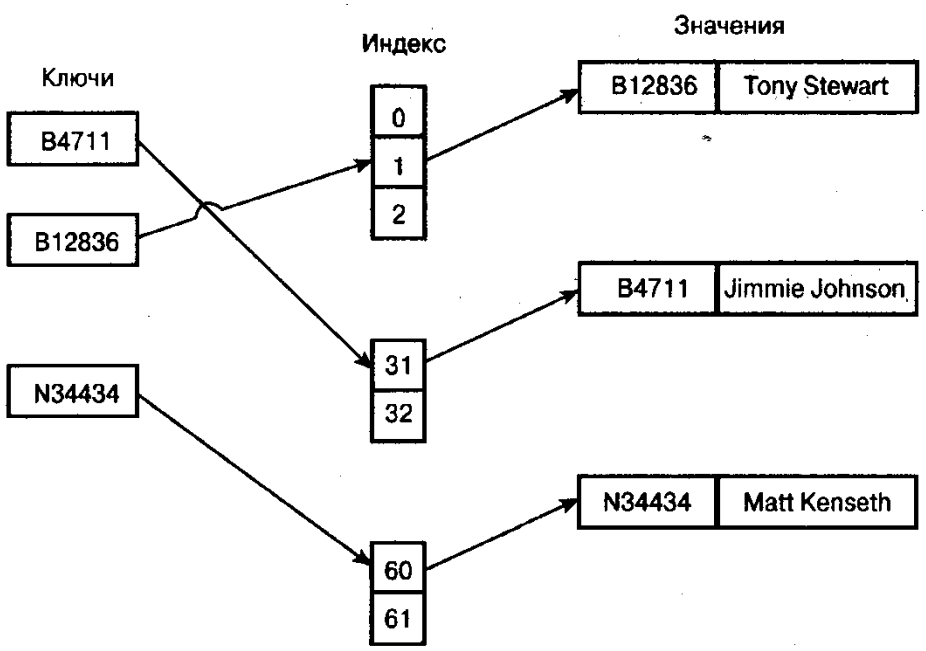

Рисунок 2.5 – Упрощенная модель словаря

Реализация метода `GetHashCode()` должна удовлетворять перечисленным ниже требованиям.

Один и тот же объект должен всегда возвращать одно и то же значение.

Разные объекты могут возвращать одно и то же значение.

Он должен выполняться насколько возможно быстро, не требуя значительных вычислительных затрат.

Он не должен генерировать исключений.

Он должен использовать как минимум одно поле экземпляра.

Значения хеш-кода должны распределяться равномерно по всему диапазону чисел, которые может хранить `int`.

Хеш-код не должен изменяться на протяжении времени существования объекта.

Чем вызвана необходимость равномерного распределения значений хеш-кода по диапазону целых чисел? Если два ключа возвращают хеш-значения, дающие один и тот же индекс, класс словаря вынужден искать ближайшее доступное свободное место для сохранения второго элемента, к тому же ему придется выполнять некоторый поиск, чтобы впоследствии извлечь требуемое значение. Понятно, что это наносит ущерб производительности, и если множество ключей дают одни и те же индексы, куда их следует поместить, вероятность конфликтов значительно возрастает. Однако благодаря способу, которым работает часть алгоритма, принадлежащая Microsoft, риск снижается до минимума, когда вычисляемое значение хеш-кода равномерно распределено между `int.MinValue` и `int.MaxValue`.

Помимо реализации `GetHashCode()` тип ключа также должен реализовывать метод `IEquatable<T>.Equals()` либо переопределять метод `Equals()` класса `Object`. Поскольку разные объекты ключа могут возвращать один и тот же хеш-код, метод `Equals()` используется при сравнении ключей словаря. Словарь проверяет два ключа `A` и `B` на эквивалентность, вызывая `A.Equals(B)`. Это означает, что потребуется обеспечить истинность следующего утверждения:

Если истинно `A.Equals(B)`, значит, `A.GetHashCode()` и `B.GetHashCode()` всегда должны возвращать один и тот же хеш-код.

Возможно, это покажется довольно тонким моментом, но это чрезвычайно важно. Ес-

ли вы придумаете такой способ переопределения этих методов, что приведенное утверждение будет истинным не всегда, то словарь, использующий экземпляры такого класса в качестве ключей, просто не будет правильно работать. Вместо этого вы столкнетесь с неприятными сюрпризами. Например, после помещения объекта в словарь вы обнаружите, что не можете его извлечь, либо при попытке извлечь элемент получите не тот, который нужно.

> *По этой причине компилятор* C# *будет отображать предупреждение, если вы переопределите* `Equals()`*, но не предоставите переопределения* `GetHashCode()`.

Для `System.Object` это условие истинно, поскольку `Equals()` просто сравнивает ссылки, а `GetHashCode()` в действительности возвращает хеш-код, основанный исключительно на адресе объекта. Это означает, что хеш-таблицы, основанные на ключе, не переопределяющем эти методы, будут работать корректно. Однако проблема такого подхода заключается в том, что ключи трактуются как эквивалентные только в том случае, если они представляют один и тот же объект. Это значит, что после помещения объекта в словарь потребуется обращаться к ссылке на ключ. Просто создать позднее экземпляр другого объекта ключа с тем же значением не получится. Если не переопределить `Equals()` и `GetHashCode()`, то тип будет не слишком удобным для использования в словаре.

Между прочим, `System.String` реализует интерфейс `IEquatable` и соответственно переопределяет `GetHashCode()`. Метод `Equals()` обеспечивает сравнение значений, а `GetHashCode()` возвращает хеш-код, основанный на значении строки. Строки с успехом могут использоваться в качестве ключей в словарях.

Числовые типы, такие как `Int32`, также реализуют интерфейс `IEquatable` и перегружают `GetHashCode()`. Однако хеш-код, возвращаемый этими типами, просто отображается на значение. Если число, которое вы хотите использовать в качестве ключа, само по себе не распределено по всему диапазону возможных целочисленных значений, применение целых в качестве ключей не отвечает правилу равномерного распределения ключевых значений для достижения наилучшей производительности. Тип `Int32` не предназначен для применения в словаре.

Если нужно использовать тип ключа, который не реализует `IEquatable` и не переопределяет `GetHashCode` соответственно значениям ключа, сохраняемым в словаре, то можно создать компаратор, реализующий интерфейс `IEqualityComparer<T>`. Интерфейс `IEqualityComparer<T>` определяет методы `GetHashCode()` и `Equals()` с аргументом – переданным объектом, так что можно предоставить реализацию, отличающуюся от типа самого объекта. Перегрузка конструктора `Dictionary<TKey, TValue>` позволяет передать объект, реализующий `IEqualityComparer<T>`. Если такой объект присвоен словарю, этот класс используется для генерации хеш-кодов и сравнения ключей.

2.7.2 Пример словаря

Рассмотрим пример использования словаря в программе, которая поддерживает словарь сотрудников компании. Словарь проиндексирован объектами `EmployeeId`, и каждый элемент, помещенный в словарь, является объектом `Employee`, хранящим подробную информацию о сотруднике.

Структура `EmployeeId` реализована для определения ключа, используемого словарем. Членами структуры являются префиксный символ и количество сотрудников. Обе эти переменных доступны только для чтения и могут быть инициализированы только в конструкторе. Ключ внутри словаря не должен изменяться, и таким образом это может быть гарантировано. Поля заполняются в конструкторе. Метод `ToString()` перегружен для получения строкового представления идентификатора сотрудника. Как того требует тип ключа, `EmployeeId` реализует интерфейс `IEquatable` и перегружает метод

GetHashCode().

```
     [Serializable]
     public class EmployeeIdException : Exception
     {
        public EmployeeIdException(string message):base(message) {
}
     }
     [Serializable]
     public struct EmployeeId : IEquatable < EmployeeId >
     {
        private readonly char prefix;
        private readonly int number;
        public EmployeeId(string id)
        {
           if (id == null) throw new ArgumentNullException("id”);
           prefix = (id.ToUpper())[0];
           int numLength = id.Length 1;
           try
           {
              number = int.Parse(id.Substring(1, numLength > 6 ? 6
: numLength));
           }
           catch (FormatException)
           {
              throw new EmployeeIdException("Invalid Employeeld
format");
           }
        }
        public override string ToString()
        {
           return prefix.ToString() + string.Format("{0, 6:000000)
", number);
        }
        public override int GetHashCode()
        {
           return (number ^ number << 16) * 0x15051505;
        }
        public bool Equals(EmployeeId other)
        {
           if (other == null) return false;
           return (prefix == other.prefix && number == oth-
er.number);
        }
        public override bool Equals(object obj)
        {
           return Equals((EmployeeId)obj);
        }
        public static bool operator == (EmployeeId left,EmployeeId
right)
        {
```

```
            return left.Equals(right);
         }
         public static bool operator != (EmployeeId left,EmployeeId
right)
         {
            return !(left == right);
         }
      }
```

Метод `Equals()`, определенный интерфейсом `IEquatable<T>`, сравнивает значения двух объектов `EmployeeId` и возвращает `true`, если значения одинаковы. Вместо реализации метода `Equals()` из интерфейса `IEquatable<T>` можно просто переопределить метод `Equals()` класса `Object`:

```
public bool Equals(EmployeeId other)
{
   if (other == null) return false;
   return (prefix == other.prefix && number == other.number);
}
```

Благодаря числовой переменной, для сотрудников ожидаются значения от 1 до 190 000. Это не заполняет всего диапазона целых чисел. Алгоритм, используемый `GetHashCode()`, сдвигает число на 16 бит влево, затем применяет операцию “исключающее ИЛИ” (XOR) к исходному числу и, наконец, умножает результат на шестнадцатеричное значение 15051505. Хеш-код достаточно равномерно распределен по диапазону целых чисел.

```
public override int GetHashCode()
{
   return (number ^ number << 16) * 0x15051505;
}
```

> *В Интернете можно найти множество более сложных алгоритмов, которые обеспечивают лучшее распределение по диапазону целых. Можно также использовать метод `GetHashCode()` строки для возврата целого.*

Класс `Employee` – это простой сущностный класс, содержащий имя, зарплату и идентификатор сотрудника. Конструктор инициализирует все значения, а метод `ToString()` возвращает строковое представление экземпляра. Реализация `ToString()` использует форматную строку для создания строкового представления из соображений производительности.

```
[Serializable]
public class Employee
{
   private string name;
   private decimal salary;
   private readonly EmployeeId id;
   public Employee(EmployeeId id, string name, decimal sala-
ry)
   {
      this.id = id;
      this.name = name;
      this.salary = salary;
```

```
        }
        public override string ToString()
        {
           return String.Format("{0} : {1, -20} {2:C}",
                id.ToString(), name, salary);
        }
     }
```

В методе `Main()` приложения-примера создается новый экземпляр класса `Dictionary<TKey, TValue>`, где ключ имеет тип `Empolyeeld`**,** а значение – тип `Empolyee`**.** Конструктор выделяет место для 31 элемента. Напомним, что емкость выражается простыми числами. Однако когда вы присваиваете значение, не являющееся простым числом, переживать не нужно. Для выделения нужной емкости сам класс `Dictionary<TKey, TValue>` находит ближайшее простое число, которое больше целого, переданного в качестве аргумента конструктора. Объекты, описывающие сотрудников и их идентификаторы, создаются и добавляются в словарь с помощью метода `Add()`. Вместо применения метода `Add()` можно также использовать индексатор для добавления ключей и значений к словарю, как показано ниже для сотрудников Dale и Jeff.

```
     static void Main()
     {
        var employees = new Dictionary <EmployeeId, Employee>
(31);
        var idKyle = new EmployeeId("T3755");
        var kyle = new Employee(idKyle, "Kyle Bush", 5443890.00m);
        employees.Add(idKyle, kyle);
        Console.WriteLine(kyle) ;

        var idCarl = new EmployeeId("F3547");
        var carl = new Employee(idCarl, "Carl Edwards",
5597120.00m);
        employees.Add(idCarl, carl);
        Console.WriteLine(carl);

        var idJimmie = new EmployeeId("C3386");
        var jimmie = new Employee(idJimmie,"Jimmie John-
son",5024710.00m);
        employees.Add(idJimmie, jimmie);
        Console.WriteLine(jimmie);

        var idDale = new EmployeeId("C3323");
        var dale = new Employee(idDale,"Dale Earnhardt Jr.",
3522740.00m);
        employees[idDale] = dale;
        Console.WriteLine(dale) ;

        var idJeff = new EmployeeId("C3234");
        var jeff = new Employee(idJeff, "Jeff Burton",
3879540.00m);
        employees[idJeff] = jeff;
        Console.WriteLine(jeff);
```

После того, как записи о сотрудниках добавлены в словарь, внутри цикла `while` они читаются из словаря. У пользователя запрашивается номер сотрудника для сохранения его в

переменной `userInput`**.** Введя `X`, пользователь может выйти из приложения. Если ключ содержится в словаре, он проверяется методом `TryGetValue()` класса `Dictionary<Tkey, TValue>`**.** Метод `TryGetValue()` возвращает `true`**,** если ключ найден, и `false` в противном случае. Если значение найдено, то значение, ассоциированное с ключом, сохраняется в переменной сотрудника. Затем оно выводится на консоль.

> *Для обращения к значению, хранящемуся в словаре, вместо `TryGetValue()` можно также использовать индексатор класса `Dictionary<Tkey, TValue>`. Однако если ключ не найден, индексатор генерирует исключение типа `KeyNotFoundException`.*

```
while (true)
{
   Console.Write("Введите идентификатор сотрудника(X для выхода)> ");
   var userInput = Console.ReadLine();
   userInput = userInput.ToUpper();
   if (userInput == "X") break;
   EmployeeId id;
   try
   {
      id = new EmployeeId(userInput);
      Employee employee;
      if (!employees.TryGetValue (id, out employee))
      {
         Console.WriteLine("Сотрудник с идентификатором {0} не существует ", id);
      else
      {
         Console.WriteLine(employee);
      }
   }
   catch (EmployeeIdException ex)
   {
      Console.WriteLine(ex.Message);
   }
  }
 }
}
```

Запуск этого приложения дает следующий вывод:

```
Введите идентификатор сотрудника (X для выхода)> C3386
C003386: Jimmie Johnson ? 5.024.710,00
Введите идентификатор сотрудника (X для выхода)> F3547
F003547 : Carl Edwards ? 5.597.120,00
Введите идентификатор сотрудника (X для выхода)> X
Для продолжения нажмите любую клавишу...
```

2.7.3 Списки поиска

`Dictionary<TKey, TValue>` поддерживает только по одному значению на ключ. Класс `Lookup<TKey, TElement>` похож на `Dictionary<TKey, TValue>`**,** но отоб-

ражает ключи на коллекцию значений. Этот класс реализован в сборке System.Core и определен в пространстве имен System.Linq.

Lookup<TKey, TElement> не может быть создан подобно обычному словарю. Вместо этого должен вызываться метод ToLookup(), который возвращает объект Lookup<TKey, TElement>. Метод ToLookup() является расширяющим методом, который доступен с каждым классом, реализующим интерфейс IEnumerable<T>. В следующем примере заполняется список объектов Racer. Поскольку List<T> реализует IEnumerable<T>, метод ToLookup() может быть вызван на списке гонщиков. Этот метод требует делегата типа Func<TSource, TKey>, определяющего селектор ключа. Здесь гонщики выбираются на основе страны с применением лямбда-выражения r => r.Country. Цикл foreach обращается только к гонщикам из Австралии (Australia), используя индексатор.

```
var racers = new List<Racer>() ;
racers.Add(new Racer("Jacques", "Villeneuve", "Canada", 11));
racers.Add(new Racer("Alan", "Jones", "Australia", 12));
racers.Add(new Racer("Jackie", "Stewart", "United Kingdom",
27));
racers.Add(new Racer("James", "Hunt", "United Kingdom", 10));
racers.Add(new Racer("Jack", "Brabham", "Australia", 14));
var lookupRacers = racers.ToLookup (r => r.Country);
foreach (Racer r in lookupRacers["Australia"])
{
   Console.WriteLine(r);
}
```

```
В выводе отображаются гонщики из Австралии:

Alan Jones
Jack Brabham
```

2.8 Сортированный словарь

Класс SortedDictioriary<TKey, Tvalue> представляет дерево бинарного поиска, в котором все элементы отсортированы на основе ключа. Тип ключа должен реализовать интерфейс IComparable<TKey>. Если тип ключа не сортируемый, компаратор можно также создать, реализовав IComparer<TKey> и указав его в качестве аргумента конструктора сортированного словаря.

Ранее в этом разделе уже упоминалось о SortedListcTKey, TValue>. Классы SortedDictionary<TKey, Tvalue> и SortedList<TKey, TValue> имеют схожую функциональность. Но поскольку SortedList<TKey, TValue> реализован в виде списка, основанного на массиве, а SortedDictionaryCTKey, Tvalue> реализован как словарь, эти классы обладают разными характеристиками.

- SortedList<TKey, TValue> использует меньше памяти, чем SortedDictionary< TKey, TValue>.
- SortedDictionary<TKey, TValue> быстрее вставляет и удаляет элементы.
- При наполнении коллекции отсортированными данными SortedListcTKey, TValue> работает быстрее, если при этом не требуется изменение емкости.

> SortedList *потребляет меньше памяти, чем* SortedDictionary. *При этом* SortedDictionary *быстрее вставляет и удаляет несортированные данные.*

2.9 Множества

Коллекция, содержащаяся только отличающиеся элементы, называется *множеством* (set). В составе .NET 4 имеются два множества – `HashSet<T>` и `SortedSet<T>`**.** Оба они реализуют интерфейс `ISet<T>`. Класс `HashSet<T>` содержит неупорядоченный список различающихся элементов, а в `SortedSet<T>` элементы упорядочены.

Интерфейс `ISet<T>` предоставляет методы для создания объединения нескольких множеств, пересечения множеств и определения, является ли одно множество надмножеством или подмножеством другого.

В следующем примере кода создается три новых множества типа `string`**,** которые заполняются названиями болидов Формулы-1. Класс `HashSet<T>` реализует интерфейс `ICollection<T>`. Однако метод `Add()` реализован явно, и как можно видеть, данный класс также предоставляет другой метод `Add()`, который отличается типом возвратам возвращается булевское значение, указывающее на то, был ли элемент добавлен. Если добавляемый элемент уже был в множестве, он не добавляется и возвращается `false`**.**

```
var companyTeams = new HashSet<string>()
     {"Ferrari", "McLaren", "Toyota", "BMW", "Renault"};
var traditionalTeams = new HashSet<string>()
     {"Ferrari", "McLaren"};
var privateTeams = new HashSet<string>()
     {"Red Bull", "Toro Rosso", "Force India", "Brawn GP"};
if(privateTeams.Add("Williams"))
   Console.WriteLine("Williams добавлен");
if (!companyTeams.Add("McLaren"))
   Console.WriteLine("McLaren уже был в множестве");
```

На консоль выводятся результаты работы двух методов **Add():**

```
Williams добавлен
McLaren уже был в множестве
```

Методы `IsSubsetOf()` и `IsSupersetOf()` сравнивают множество с коллекцией, реализующей интерфейс `IEnumerable<T>`**,** и возвращают результат булевского типа. Здесь `IsSubsetOf()` проверяет, все ли элементы. `tradionalTeams` включены в `company-Teams`**,** что верно, а `IsSupersetOf()` проверяет, содержит ли `tradionalTeams` какие-то дополнительные элементы по сравнению с `companyTeams`**.**

```
if (traditionalTeams.IsSubsetOf(companyTeams))
{
   Console.WriteLine("traditionalTeams является подмножеством
companyTeams");
}
if (companyTeams.IsSupersetOf(traditionalTeams))
{
   Console.WriteLine("companyTeams является надмножеством
traditionalTeams");
}
```

Вывод показан ниже:

```
traditionalTeams является подмножеством companyTeams
companyTeams является надмножеством traditionalTeams
```

`Williams` – также традиционная команда, и потому она включена в коллекцию

traditionalTeams:

```
traditionalTeams.Add("Williams");
if (privateTeams.Overlaps(traditionalTeams))
{
   Console.WriteLine("Как минимум, одна команда относится к " 
+ "традиционным и частным командам");
}
```

Поскольку пересечение имеет место, вывод будет таким:

```
Как минимум, одна команда относится к традиционным и частным 
командам
```

Переменная `allTeams`, ссылающаяся на `SortedSet<string>`, заполняется объединением `companyTeams`, `privateTeams` и `traditionalTeams` за счет вызова метода `UnionWith()`:

```
var allTeams = new SortedSet<string>(companyTeams);
allTeams.UnionWith(privateTeams);
allTeams.UnionWith(traditionalTeams);
Console.WriteLine();
Console.WriteLine("Все команды");
foreach (var team in allTeams)
{
   Console.WriteLine(team);
}
```

Это вернет все команды, но каждая из них будет представлена только один раз, потому что множество содержит только уникальные элементы. А поскольку применяется контейнер `SortedSet<string>`, результат упорядочен:

```
Все команды
BMW
Brawn GP
Ferrari
Force India
McLaren
Red Bull
Renault
Toro Rosso
Toyota
Williams
```

Метод `ExceptWith()` удаляет все частные команды из множества `allTeams`:

```
allTeams.ExceptWith(privateTeams);
Console.WriteLine();
Console.WriteLine("Частные команды исключены");
foreach (var team in allTeams)
{
   Console.WriteLine(var team in allTeams);
}
```

Оставшиеся элементы в коллекции не содержат частные команды:

```
Частные команды исключены
BMW
Ferrari
McLaren
Renault
Toyota
```

3. Задания

Описать структуру, соответствующую заданиям лабораторной работы 9. Создать параметризованную коллекцию для хранения описанной структуры. Вид коллекции выбрать самостоятельно. Написать консольное приложение для работы с этой коллекцией, позволяющее выполнять:

1. добавление элемента в коллекцию с клавиатуры;
2. считывание данных из файла;
3. запись данных в тот же или указанный файл;
4. сортировку данных по различным критериям;
5. поиск элемента по заданному полю;
6. вывод всех элементов, удовлетворяющих заданному условию;
7. удаление элемента из коллекции.

Приложение должно предусматривать выбор действия и обработку возможных ошибок пользователя с помощью исключений.

4 Контрольные вопросы

1. Что такое обобщенные коллекции? В каком пространстве имен объявляются обобщенные коллекции?
2. Какие существуют интерфейсы обобщенных коллекций?
3. Структура KeyValuePair<TKey, TValue>: описание, методы, свойства.
4. Стек: классы Stack и Stack<T>: описание, методы, свойства.
5. Очередь: классы Queue и Queue<T>: описание, методы, свойства.
6. Связный список: класс LinkedList<T>: описание, методы, свойства.
7. Словарь: класс Dictionary<TKey, TValue>: описание, методы, свойства.
8. Сортированный словарь: класс SortedDictioriary<TKey, Tvalue>: описание, методы, свойства.
9. Множества: классы HashSet<T> и SortedSet<T>: описание, методы, свойства.