ГЛАВА 9

Коллекции и обобщения

Любому приложению, создаваемому с помощью платформы.NET, потребуется решать проблемы поддержки и манипулирования набором значений данных в памяти. Эти значения данных могут поступать из множества местоположений, включая реляционную базу данных, локальный текстовый файл, XML-документ, вызов веб-службы и даже через предоставляемый пользователем источник ввода.

В первом выпуске платформы .NET программисты часто применяли классы из пространства имен `System.Collections` для хранения и взаимодействия с элементами данных, используемыми внутри приложения. В версии .NET 2.0 язык программирования C# был расширен для поддержки средства под названием *обобщения*; и вместе с этим изменением в библиотеках базовых классов появилось совершенно новое пространство имен: `System.Collections.Generic`.

В этой главе представлен обзор различных пространств имен и типов коллекций (обобщенных и необобщенных), находящихся в библиотеках базовых классов .NET. Как вы увидите, обобщенные контейнеры часто превосходят свои необобщенные аналоги, поскольку они обычно предоставляют лучшую безопасность к типам и преимущества в плане производительности. После объяснения того, как создавать и манипулировать обобщенными элементами внутри платформы, в оставшейся части главы будет показано, как создавать собственные методы и обобщенные типы. Вы узнаете о роли *ограничений* (*и соответствующего ключевого слова* `where` в C#), которые позволяют строить исключительно безопасные к типам классы.

Побудительные причины создания классов коллекций

Самым элементарным контейнером, который можно использовать для хранения данных приложения, является, несомненно, массив. Как было показано в главе 4, массивы C# позволяют определять наборы типизированных элементов (включая массив объектов типа `System.Object`, по сути представляющий собой массив любых типов) с фиксированным верхним пределом. Кроме того, вспомните из главы 4, что все переменные массивов C# имеют дело с функциональностью из класса `System.Array`. В качестве краткого напоминания, взгляните на следующий метод `Main()`, который создает массив текстовых данных и манипулирует его содержимым несколькими способами:

```
static void Main(string[] args)
{
  // Создать массив строковых данных.
  string[] strArray = {"First", "Second", "Third" };

  // Отобразить количество элементов в массиве с помощью свойства Length.
  Console.WriteLine("This array has {0} items.", strArray.Length);
```

```
  Console.WriteLine();

  // Отобразить содержимое массива с использованием перечислителя.
  foreach (string s in strArray)
  {
    Console.WriteLine("Array Entry: {0}", s);
  }
  Console.WriteLine();

  // Обратить массив и снова вывести его содержимое.
  Array.Reverse(strArray);
  foreach (string s in strArray)
  {
    Console.WriteLine("Array Entry: {0}", s);
  }

  Console.ReadLine();
}
```

Хотя базовые массивы могут быть удобны для управления небольшими объемами данных фиксированного размера, бывает также немало случаев, когда требуются более гибкие структуры данных, такие как динамически растущие и сокращающиеся контейнеры или контейнеры, которые хранят объекты, отвечающие только определенному критерию (например, объекты, унаследованные от заданного базового класса, или объекты, реализующие определенный интерфейс). При использовании простого массива всегда помните о том, что он имеет "фиксированный размер". Если вы создали массив из трех элементов, то вы и получите только три элемента; следовательно, приведенный ниже код даст в результате исключение времени выполнения (конкретнее — `IndexOutOfRangeException`):

```
static void Main(string[] args)
{
  // Создать массив строковых данных.
  string[] strArray = { "First", "Second", "Third" };

  // Попытаться добавить новый элемент после конца массива. Ошибка времени выполнения!
  strArray[3] = "new item?";
  ...
}
```

Чтобы помочь в преодолении ограничений простого массива, библиотеки базовых классов .NET поставляются с несколькими пространствами имен, содержащими *классы коллекций*. В отличие от простого массива C#, классы коллекций построены с возможностью динамического изменения своих размеров на лету при вставке либо удалении из них элементов. Более того, многие классы коллекций предлагают улучшенную безопасность к типам и оптимизированы для обработки содержащихся внутри данных эффективно с точки зрения расхода памяти. По мере чтения этой главы, вы быстро заметите, что класс коллекции может принадлежать к одной из двух обширных категорий:

- необобщенные коллекции (в основном находящиеся в пространстве имен `System.Collections`);
- обобщенные коллекции (в основном находящиеся в пространстве имен `System.Collections.Generic`).

Необобщенные коллекции обычно предназначены для оперирования над типами `System.Object` и, таким образом, являются слабо типизированными контейнерами (тем не менее, некоторые необобщенные коллекции работают только со специфическим типом данных, таким как объекты `string`). В противоположность этому, обобщенные коллекции являются намного более безопасными к типам, учитывая, что вы должны

указать "тип типа", который они будут содержать после создания. Как вы увидите, признаком любого обобщенного элемента является наличие "параметра типа", обозначаемого с помощью угловых скобок (например, `List<T>`). Детали обобщений (в том числе связанные с ними преимущества) будут рассматриваться позже в этой главе. А сейчас давайте ознакомимся с некоторыми ключевыми типами необобщенных коллекций из пространств имен `System.Collections` и `System.Collections.Specialized`.

Пространство имен `System.Collections`

С момента появления платформы .NET программисты часто использовали классы необобщенных коллекций из пространства имен `System.Collecitons`, которое содержит набор классов, предназначенных для управления и организации больших объемов данных в памяти. В табл. 9.1 документированы некоторые наиболее часто используемые классы коллекций, определенные в этом пространстве имен, а также основные интерфейсы, которые они реализуют.

На заметку! Любое приложение .NET, построенное с помощью .NET 2.0 или последующих версий, должно игнорировать классы в `System.Collections` и отдавать предпочтение соответствующим классам из `System.Collections.Generic`. Тем не менее, важно знать основы необобщенных классов коллекций, поскольку может возникнуть необходимость в сопровождении унаследованного программного обеспечения.

Таблица 9.1. Полезные типы из `System.Collections`

Класс `System.Collections`	Назначение	Основные реализуемые интерфейсы
`ArrayList`	Представляет коллекцию динамически изменяемого размера, содержащую объекты в определенном порядке	`IList`, `ICollection`, `IEnumerable` и `ICloneable`
`BitArray`	Управляет компактным массивом битовых значений, которые представляются как булевские, где `true` обозначает установленный (1) бит, а `false` — неустановленный (0) бит	`ICollection`, `IEnumerable` и `ICloneable`
`Hashtable`	Представляет коллекцию пар "ключ/значение", организованных на основе хеш-кода ключа	`IDictionary`, `ICollection`, `IEnumerable` и `ICloneable`
`Queue`	Представляет стандартную очередь объектов, работающую по алгоритму FIFO ("первый вошел — первый вышел")	`ICollection`, `IEnumerable` и `ICloneable`
`SortedList`	Представляет коллекцию пар "ключ/значение", отсортированных по ключу и доступных по ключу и по индексу	`IDictionary`, `ICollection`, `IEnumerable` и `ICloneable`
`Stack`	Представляет стек LIFO ("последний вошел — первый вышел"), поддерживающий функциональность заталкивания и выталкивания, а также считывания	`ICollection`, `IEnumerable` и `ICloneable`

Интерфейсы, реализованные этими классами коллекций, представляют огромное "окно" в их общую функциональность. В табл. 9.2 представлено описание общей природы этих основных интерфейсов, часть из которых поверхностно рассматривалась в главе 8.

Таблица 9.2. Основные интерфейсы, поддерживаемые классами `System.Collections`

Интерфейс `System.Collections`	Назначение
`ICollection`	Определяет общие характеристики (т.е. размер, перечисление и безопасность к потокам) всех необобщенных типов коллекций
`ICloneable`	Позволяет реализующему объекту возвращать копию самого себя вызывающему коду
`IDictionary`	Позволяет объекту необобщенной коллекции представлять свое содержимое в виде пар "имя/значение"
`IEnumerable`	Возвращает объект, реализующий интерфейс `IEnumerator` (см. следующую строку в этой таблице)
`IEnumerator`	Делает возможной итерацию в стиле `foreach` по элементам коллекции
`IList`	Обеспечивает поведение добавления, удаления и индексирования элементов в списке объектов

Иллюстративный пример: работа с `ArrayList`

Возможно, вы уже имеете первоначальный опыт использования (или реализации) некоторых из указанных выше классических структур данных, например, стеков, очередей или списков. Если это не так, то позже в главе, при рассмотрении обобщенных аналогов таких структур, будут предоставлены дополнительные сведения об отличиях между ними. А пока что взгляните на метод `Main()`, в котором используется объект `ArrayList`. Обратите внимание, что мы можем добавлять (и удалять) элементы на лету, а контейнер автоматически соответствующим образом изменяет свой размер:

```
// Для доступа к ArrayList потребуется импортировать System.Collections.
static void Main(string[] args)
{
  ArrayList strArray = new ArrayList();
  strArray.AddRange(new string[] { "First", "Second", "Third" });

  // Отобразить количество элементов в ArrayList.
  Console.WriteLine("This collection has {0} items.", strArray.Count);
  Console.WriteLine();

  // Добавить новый элемент и отобразить текущее их количество.
  strArray.Add("Fourth!");
  Console.WriteLine("This collection has {0} items.", strArray.Count);

  // Отобразить содержимое.
  foreach (string s in strArray)
  {
    Console.WriteLine("Entry: {0}", s);
  }
  Console.WriteLine();
}
```

Несложно догадаться, что класс `ArrayList` имеет множество полезных членов помимо свойства `Count` и методов `AddRange()` и `Add()`, которые подробно описаны в до-

кументации по .NET Framework. К слову, другие классы System.Collections (Stack, Queue и т.д.) также подробно документированы в справочной системе .NET.

Тем не менее, очень важно отметить, что в большинстве ваших проектов .NET, скорее всего, классы коллекций из пространства имен System.Collections использоваться не будут! В наши дни намного чаще применяются их обобщенные аналоги, расположенные в пространстве имен System.Collections.Generic. Учитывая это, остальные необобщенные классы из System.Collections здесь не обсуждаются (и примеры их использования не приводятся).

Обзор пространства имен System.Collections.Specialized

System.Collections — не единственное пространство имен .NET, которое содержит необобщенные классы коллекций. Например, в пространстве имен System.Collections.Specialized определено несколько специализированных типов коллекций. В табл. 9.3 описаны некоторые наиболее полезные типы в этом конкретном пространстве имен, причем все они необобщенные.

Таблица 9.3. Полезные классы System.Collections.Specialized

Тип System.Collections.Specialized	Назначение
HybridDictionary	Этот класс реализует интерфейс IDictionary за счет использования ListDictionary, когда коллекция маленькая, и затем переключается на Hashtable, когда коллекция становится большой
ListDictionary	Этот класс удобен, когда необходимо управлять небольшим количеством элементов (10 или около того), которые могут изменяться со временем. Для управления своими данными класс использует односвязный список
StringCollection	Этот класс обеспечивает оптимальный способ для управления крупными коллекциями строковых данных
BitVector32	Этот класс предоставляет простую структуру, которая хранит булевские значения и небольшие целые числа в 32 битах памяти

Помимо указанных конкретных типов это пространство имен также содержит множество дополнительных интерфейсов и абстрактных базовых классов, которые можно применять в качестве стартовых точек для создания специальных классов коллекций. Хотя эти “специализированные” типы и могут оказаться тем, что требуется для ваших проектов в ряде ситуаций, здесь они рассматриваться не будут. Опять-таки, во многих случаях вы, скорее всего, обнаружите, что пространство имен System.Collections.Generic предлагает классы с похожей функциональностью, но с набором преимуществ.

На заметку! В библиотеках базовых классов .NET доступны два дополнительных пространства имен, связанных с коллекциями (System.Collections.ObjectModel и System.Collections.Concurrent). Первое из них будет описано позже в этой главе, когда вы освоите тему обобщений. Пространство имен System.Collections.Concurrent предоставляет класс коллекций, безопасные к потокам (многопоточность рассматривается в главе 19).

Проблемы, связанные с необобщенными коллекциями

Хотя на протяжении многих лет с применением этих необобщенных классов коллекций (и интерфейсов) было построено немало успешных приложений .NET, опыт показал, что применение этих типов может быть сопряжено с множеством проблем.

Первая проблема состоит в том, что использование классов коллекций `System.Collections` и `System.Collections.Specialized` приводит к созданию низкопроизводительного кода, особенно в случае манипуляций с числовыми данными (т.е. типами значений). Как вскоре будет показано, при хранении таких данных в любом необобщенном классе коллекции, прототипированном для работы с `System.Object`, среде CLR приходится выполнять массу операций перемещения данных в памяти, что может значительно снизить скорость выполнения.

Вторая проблема связана с тем, что большинство необобщенных классов коллекций не являются безопасными к типам, т.к. они были созданы для оперирования на `System.Object` и потому могут содержать в себе все что угодно. Если разработчику .NET требовалось создать безопасную в отношении типов коллекцию (т.е. контейнер, который может содержать объекты, реализующие только определенный интерфейс), то единственным реальным вариантом было создание совершенно нового класса коллекции собственноручно. Это не слишком трудоемкая задача, но довольно утомительная.

Прежде чем будет показано, как использовать обобщения в своих программах, стоит глубже рассмотреть недостатки необобщенных классов коллекций; это поможет лучше понять проблемы, которые был призван решить механизм обобщений. Давайте создадим новое консольное приложение по имени `IssuesWithNongenericCollections` и затем импортируем пространство имен `System.Collections` в начале кода C#:

```
using System.Collections;
```

Проблема производительности

Как уже должно быть известно из главы 4, платформа .NET поддерживает две обширных категории данных: типы значений и ссылочные типы. Поскольку в .NET определены две основных категории типов, однажды может возникнуть необходимость представить переменную одной категории в виде переменной другой категории. Для этого в C# предлагается простой механизм, называемый *упаковкой* (boxing), который служит для сохранения данных типа значения в ссылочной переменной. Предположим, что в методе по имени `SimpleBoxUnboxOperation()` создана локальная переменная типа `int`. Если далее в приложении понадобится представить этот тип значения в виде ссылочного типа, значение следует *упаковать*, как показано ниже:

```
private static void SimpleBoxUnboxOperation()
{
  // Создать переменную ValueType (int).
  int myInt = 25;

  // Упаковать int в ссылку на object.
  object boxedInt = myInt;
}
```

Упаковку можно формально определить как процесс явного присваивания типа значения переменной `System.Object`. При упаковке значения среда CLR размещает в куче новый объект и копирует значение типа значения (в данном случае `25`) в этот экземпляр. В качестве результата возвращается ссылка на вновь размещенный в куче объект.

Противоположная операция также разрешена, и она называется *распаковкой* (unboxing). Распаковка — это процесс преобразования значения, хранящегося в объектной ссылке, обратно в соответствующий тип значения в стеке. Синтаксически операция распаковки выглядит как нормальная операция приведения, однако ее семантика несколько отличается. Среда CLR начинает с проверки того, что полученный тип данных эквивалентен упакованному типу, и если это так, то копирует значение обратно в находящуюся в стеке переменную. Например, следующие операции распаковки работают успешно при условии, что типом `boxedInt` в действительности является `int`:

```
private static void SimpleBoxUnboxOperation()
{
  // Создать переменную ValueType (int).
  int myInt = 25;

  // Упаковать int в ссылку на object.
  object boxedInt = myInt;

  // Распаковать ссылку обратно в int.
  int unboxedInt = (int)boxedInt;
}
```

Когда компилятор C# встречает синтаксис упаковки/распаковки, он генерирует CIL-код, содержащий коды операций `box/unbox`. Заглянув в сборку с помощью утилиты `ildasm.exe`, можно найти там следующий CIL-код:

```
.method private hidebysig static void SimpleBoxUnboxOperation() cil managed
{
  // Code size 19 (0x13)
  .maxstack 1
  .locals init ([0] int32 myInt, [1] object boxedInt, [2] int32 unboxedInt)
  IL_0000:  nop
  IL_0001:  ldc.i4.s 25
  IL_0003:  stloc.0
  IL_0004:  ldloc.0
  IL_0005:  box [mscorlib]System.Int32
  IL_000a:  stloc.1
  IL_000b:  ldloc.1
  IL_000c:  unbox.any [mscorlib]System.Int32
  IL_0011:  stloc.2
  IL_0012:  ret
} // end of method Program::SimpleBoxUnboxOperation
```

Помните, что в отличие от обычного приведения распаковка *должна* производиться только в соответствующий тип данных. Попытка распаковать порцию данных в некорректный тип данных приводит к генерации исключения `InvalidCastException`. Для полной безопасности следовало бы поместить каждую операцию распаковки в конструкцию `try/catch`, однако делать это для абсолютно каждой операции распаковки в приложении может оказаться довольно трудоемкой задачей. Взгляните на следующий измененный код, который выдаст ошибку, поскольку предпринята попытка распаковать упакованный `int` в `long`:

```
static void SimpleBoxUnboxOperation()
{
  // Создать переменную ValueType (int).
  int myInt = 25;

  // Упаковать int в ссылку на object.
  object boxedInt = myInt;

  // Распаковать в неверный тип данных, чтобы
  // инициировать исключение времени выполнения.
```

```
  try
  {
    long unboxedInt = (long)boxedInt;
  }
  catch (InvalidCastException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

На первый взгляд упаковка/распаковка может показаться довольно несущественным средством языка, представляющим скорее академический интерес, нежели практическую ценность. На самом деле процесс упаковки/распаковки очень полезен, поскольку позволяет предположить, что все можно трактовать как `System.Object`, причем CLR берет на себя все заботы о деталях, связанных с памятью.

Давайте посмотрим на практическое применение этих приемов. Предположим, что создан необобщенный класс `System.Collections.ArrayList` для хранения множества числовых (расположенных в стеке) данных. Члены `ArrayList` прототипированы для работы с данными `System.Object`. Теперь рассмотрим методы `Add()`, `Insert()`, `Remove()`, а также индексатор класса:

```
public class ArrayList : object,
  IList, ICollection, IEnumerable, ICloneable
{
  ...
  public virtual int Add(object value);
  public virtual void Insert(int index, object value);
  public virtual void Remove(object obj);
  public virtual object this[int index] { get; set; }
}
```

Класс `ArrayList` ориентирован на работу с экземплярами `object`, которые представляют данные, расположенные в куче, поэтому может показаться странным, что следующий код компилируется и выполняется без ошибок:

```
static void WorkWithArrayList()
{
  // Типы значений упаковываются автоматически
  // при передаче методу, запросившему объект.
  ArrayList myInts = new ArrayList();
  myInts.Add(10);
  myInts.Add(20);
  myInts.Add(35);
}
```

Несмотря на непосредственную передачу числовых данных в методы, требующие тип `object`, исполняющая среда автоматически упаковывает их в данные, расположенные в стеке. При последующем извлечении элемента из `ArrayList` с использованием индексатора типа потребуется распаковать посредством операции приведения объект, находящийся в куче, в целочисленное значение, расположенное в стеке. Помните, что индексатор `ArrayList` возвращает `System.Object`, а не `System.Int32`:

```
static void WorkWithArrayList()
{
  // Типы значений автоматически упаковываются, когда
  // передаются члену, принимающему объект.
  ArrayList myInts = new ArrayList();
  myInts.Add(10);
  myInts.Add(20);
  myInts.Add(35);
```

```
    // Распаковка происходит, когда объект преобразуется
    // обратно в расположенные в стеке данные.
    int i = (int)myInts[0];

    // Теперь значение вновь упаковывается, т.к. WriteLine() требует объектные типы!
    Console.WriteLine("Value of your int: {0}", i);
  }
```

Обратите внимание, что расположенные в стеке значения `System.Int32` упаковываются перед вызовом `ArrayList.Add()`, чтобы их можно было передать в требуемом виде `System.Object`. Также отметьте, что объекты `System.Object` распаковываются обратно в `System.Int32` после их извлечения из `ArrayList` через операцию приведения только для того, чтобы вновь быть упакованными для передачи в метод `Console.WriteLine()`, поскольку этот метод оперирует переменными `System.Object`.

Хотя упаковка и распаковка очень удобны с точки зрения программиста, этот упрощенный подход к передаче данных между стеком и кучей влечет за собой проблемы, связанные с производительностью (это касается как скорости выполнения, так и размера кода), а также недостаток безопасности к типам. Чтобы понять, в чем состоят проблемы с производительностью, взгляните на перечень действий, которые должны быть выполнены при упаковке и распаковке простого целого числа.

1. Новый объект должен быть размещен в управляемой куче.
2. Значение данных, находящихся в стеке, должно быть передано в выделенное место в памяти.
3. При распаковке значение, которое хранится в объекте, находящемся в куче, должно быть передано обратно в стек.
4. Неиспользуемый больше объект в куче будет (в конечном итоге) удален сборщиком мусора.

Хотя существующий метод `Main()` не является основным узким местом в смысле производительности, вы определенно это почувствуете, если `ArrayList` будет содержать тысячи целочисленных значений, к которым программа обращается на регулярной основе. В идеальном случае хотелось бы манипулировать расположенными в стеке данными внутри контейнера, не имея проблем с производительностью. Было бы хорошо иметь возможность извлекать данные из контейнера, обходясь без конструкций `try/catch` (именно это обеспечивают обобщения).

Проблемы с безопасностью типов

Проблема безопасности типов уже затрагивалась, когда речь шла об операциях распаковки. Вспомните, что данные должны быть распакованы в тот же тип, который был для них объявлен перед упаковкой. Однако существует и другой аспект безопасности типов, который следует иметь в виду в мире без обобщений: тот факт, что классы из `System.Collections` могут хранить все что угодно, поскольку их члены прототипированы для работы с `System.Object`. Например, в следующем методе контейнер `ArrayList` хранит произвольные фрагменты несвязанных данных:

```
static void ArrayListOfRandomObjects()
{
  // ArrayList может хранить все что угодно.
  ArrayList allMyObject = new ArrayList();
  allMyObjects.Add(true);
  allMyObjects.Add(new OperatingSystem(PlatformID.MacOSX, new Version(10, 0)));
  allMyObjects.Add(66);
  allMyObjects.Add(3.14);
}
```

В некоторых случаях действительно необходим исключительно гибкий контейнер, который может хранить буквально все. Однако в большинстве ситуаций понадобится *безопасный в отношении типов* контейнер, который может оперировать только определенным типом данных, например, контейнер, который хранит только подключения к базе данных, битовыми образами или объекты, совместимые с `IPointy`.

До появления обобщений единственным способом решения этой проблемы было создание вручную класса строго типизированной коллекции. Предположим, что создана специальная коллекция, которая может содержать только объекты типа `Person`:

```
public class Person
{
  public int Age {get; set;}
  public string FirstName {get; set;}
  public string LastName {get; set;}

  public Person(){}
  public Person(string firstName, string lastName, int age)
  {
    Age = age;
    FirstName = firstName;
    LastName = lastName;
  }
  public override string ToString()
  {
    return string.Format("Name: {0} {1}, Age: {2}",
      FirstName, LastName, Age);
  }
}
```

Чтобы построить *коллекцию, позволяющую хранить только объекты* `Person`, можно определить переменную-член `System.Collection.ArrayList` внутри класса по имени `PeopleCollection` и сконфигурировать все члены для работы со строго типизированными объектами `Person` вместо объектов типа `System.Object`. Ниже приведен простой пример (реальная коллекция производственного уровня должна включать множество дополнительных членов и расширять абстрактный базовый класс из пространства имен `System.Collections` или `System.Collections.Specialized`):

```
public class PeopleCollection : IEnumerable
{
  private ArrayList arPeople = new ArrayList();

  // Приведение для вызывающего кода.
  public Person GetPerson(int pos)
  { return (Person)arPeople[pos]; }

  // Вставка только объектов Person.
  public void AddPerson(Person p)
  { arPeople.Add(p); }

  public void ClearPeople()
  { arPeople.Clear(); }

  public int Count
  { get { return arPeople.Count; } }

  // Поддержка перечисления с помощью foreach.
  IEnumerator IEnumerable.GetEnumerator()
  { return arPeople.GetEnumerator(); }
}
```

Обратите внимание, что класс PeopleCollection реализует интерфейс IEnumerable, который делает возможной итерацию в стиле foreach по всем содержащимся в коллекции элементам. Кроме того, методы GetPerson() и AddPerson() прототипированы на работу только с объектами Person, а не битовыми образами, строками, подключениями к базе данных или другими элементами. За счет создания таких классов обеспечивается безопасность типов, учитывая, что компилятор C# будет иметь возможность выявить любую попытку вставки элемента несовместимого типа:

```
static void UsePersonCollection()
{
  Console.WriteLine("***** Custom Person Collection *****\n");
  PersonCollection myPeople = new PersonCollection();
  myPeople.AddPerson(new Person("Homer", "Simpson", 40));
  myPeople.AddPerson(new Person("Marge", "Simpson", 38));
  myPeople.AddPerson(new Person("Lisa", "Simpson", 9));
  myPeople.AddPerson(new Person("Bart", "Simpson", 7));
  myPeople.AddPerson(new Person("Maggie", "Simpson", 2));

  // Это вызовет ошибку при компиляции!
  // myPeople.AddPerson(new Car());

  foreach (Person p in myPeople)
    Console.WriteLine(p);
}
```

Хотя подобные специальные коллекции гарантируют безопасность типов, такой подход все же обязывает создавать (в основном идентичные) специальные коллекции для каждого уникального типа данных, который планируется хранить. Таким образом, если нужна специальная коллекция, которая будет способна оперировать только классами, унаследованными от базового класса Car, понадобится построить очень похожий класс коллекции:

```
public class CarCollection : IEnumerable
{
  private ArrayList arCars = new ArrayList();

  // Приведение для вызывающего кода.
  public Car GetCar(int pos)
  { return (Car) arCars[pos]; }

  // Вставка только объектов Car.
  public void AddCar(Car c)
  { arCars.Add(c); }

  public void ClearCars()
  { arCars.Clear(); }

  public int Count
  { get { return arCars.Count; } }

  // Поддержка перечисления с помощью foreach.
  IEnumerator IEnumerable.GetEnumerator()
  { return arCars.GetEnumerator(); }
}
```

Однако эти специальные контейнеры мало помогают в решении проблем упаковки/распаковки. Даже если создать специальную коллекцию по имени IntCollection, предназначенную для работы только с элементами System.Int32, все равно придется выделить некоторый тип объекта для хранения данных (например, System.Array и ArrayList):

```
public class IntCollection : IEnumerable
{
  private ArrayList arInts = new ArrayList();

  // Получить int (выполнить распаковку).
  public int GetInt(int pos)
  { return (int)arInts[pos]; }

  // Вставить int (выполнить упаковку).
  public void AddInt(int i)
  { arInts.Add(i); }

  public void ClearInts()
  { arInts.Clear(); }

  public int Count
  { get { return arInts.Count; } }

  IEnumerator IEnumerable.GetEnumerator()
  { return arInts.GetEnumerator(); }
}
```

Независимо от того, какой тип выбран для хранения целых чисел, дилеммы упаковки нельзя избежать, применяя необобщенные контейнеры.

Первый взгляд на обобщенные коллекции

В случае использования классов обобщенных коллекций исчезают все описанные выше проблемы, включая затраты на упаковку/распаковку и недостаток безопасности типов. Кроме того, потребность в создании специального класса (обобщенной) коллекции становится довольно редкой. Вместо построения специальных коллекций, которые могут хранить людей, автомобили и целые числа, можно обратиться к обобщенному классу коллекции и указать тип хранимых элементов.

В показанном ниже методе класс `List<T>` (из пространства имен `System.Collection.Generic`) используется для хранения различных типов данных в строго типизированной манере (пока не обращайте внимания на детали синтаксиса обобщений):

```
static void UseGenericList()
{
  Console.WriteLine("***** Fun with Generics *****\n");

  // Этот List<> может хранить только объекты Person.
  List<Person> morePeople = new List<Person>();
  morePeople.Add(new Person ("Frank", "Black", 50));
  Console.WriteLine(morePeople[0]);

  // Этот List<> может хранить только целые числа.
  List<int> moreInts = new List<int>();
  moreInts.Add(10);
  moreInts.Add(2);
  int sum = moreInts[0] + moreInts[1];

  // Ошибка компиляции! Объект Person не может быть добавлен в список int!
  // moreInts.Add(new Person());
}
```

Первая коллекция `List<T>` может содержать только объекты `Person`. Поэтому выполнять приведение при извлечении элементов из контейнера не требуется, что делает этот подход более безопасным в отношении типов. Вторая коллекция `List<T>` может хранить только целые числа, и все они размещены в стеке; другими словами, здесь не происходит никакой скрытой упаковки/распаковки, как это имеет место в необобщенном `ArrayList`.

Ниже приведен краткий перечень преимуществ обобщенных контейнеров по сравнению с их необобщенными аналогами.

- Обобщения обеспечивают более высокую производительность, поскольку не страдают от проблем упаковки/распаковки при хранении типов значений.
- Обобщения являются безопасными в отношении типов, т.к. могут содержать только объекты указанного типа.
- Обобщения значительно сокращают потребность в специальных типах коллекций, потому что вы указываете "тип типа" при создании обобщенного контейнера.

Исходный код. Проект `IssuesWithNonGenericCollections` доступен в подкаталоге `Chapter 09`.

Роль параметров обобщенных типов

Обобщенные классы, интерфейсы, структуры и делегаты буквально разбросаны по всей базовой библиотеке классов .NET, и они могут быть частью любого пространства имен .NET. Кроме того, учтите, что использование обобщений далеко не ограничивается одним лишь определением класса коллекции. Разумеется, в оставшейся части книги вы увидите много других обобщений, применяемых для разных целей.

На заметку! Обобщенными могут быть только классы, структуры, интерфейсы и делегаты, но не перечисления.

Отличить обобщенный элемент в документации .NET Framework или браузере объектов Visual Studio от других элементов очень легко по наличию пары угловых скобок с буквой или другой лексемой. На рис. 9.1 показан браузер объектов Visual Studio, который отображает множество обобщенных элементов из пространства имен `System.Collections.Generic`, включая выделенный класс `List<T>`.

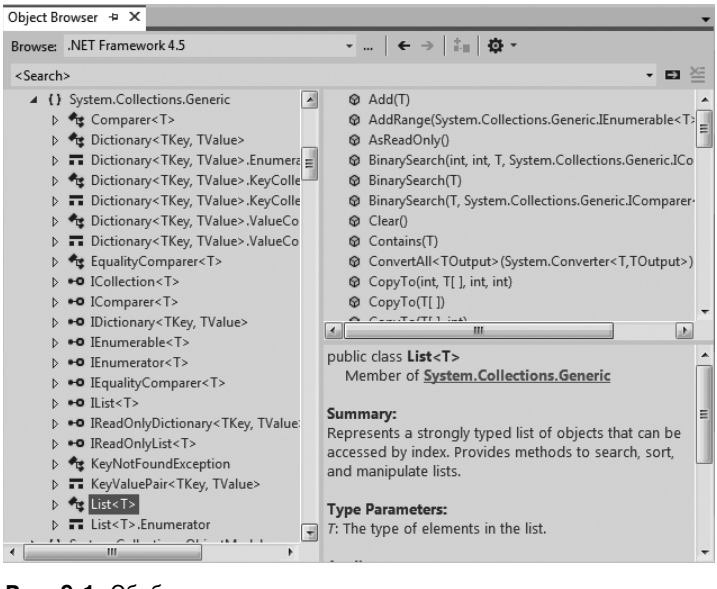

Рис. 9.1. Обобщенные элементы, поддерживающие параметры типа

Формально эти лексемы можно называть *параметрами типа*, однако в более дружественных к пользователю терминах их можно считать просто *заполнителями*. Конструкцию <T> можно воспринимать как *типа T*. Таким образом, IEnumerable<T> можно читать как *IEnumerable типа T*, или, говоря иначе, *перечисление типа T*.

На заметку! Имя параметра типа (заполнитель) не важно, и это — дело вкуса разработчика, создавшего обобщенный элемент. Тем не менее, обычно для представления типов используется T, для представления ключей — TKey или K, а для представления значений — TValue или V.

При создании обобщенного объекта, реализации обобщенного интерфейса или вызове обобщенного члена должно быть указано значение для параметра типа. Как в этой главе, так и в остальной части книги будет продемонстрировано немало примеров. Однако для начала следует ознакомиться с основами взаимодействия с обобщенными типами и членами.

Указание параметров типа для обобщенных классов и структур

При создании экземпляра обобщенного класса или структуры параметр типа указывается, когда объявляется переменная и когда вызывается конструктор. В предыдущем фрагменте кода было показано, что UseGenericList() определяет два объекта List<T>:

```
// Этот List<> может хранить только объекты Person.
List<Person> morePeople = new List<Person>();
```

Этот фрагмент можно трактовать как *List<> объектов T, где T — тип Person*, или более просто — *список объектов персон*. После указания параметра типа обобщенного элемента его нельзя изменить (помните: обобщения предназначены для поддержки безопасности типов). Когда параметр типа задается для обобщенного класса или структуры, все вхождения заполнителей заменяются указанным значением.

Просмотрев полное объявление обобщенного класса List<T> в браузере объектов Visual Studio, можно заметить, что заполнитель T используется в определении повсеместно. Ниже приведен частичный листинг (обратите внимание на элементы, выделенные полужирным):

```
// Частичный листинг класса List<T>.
namespace System.Collections.Generic
{
  public class List<T> :
    IList<T>, ICollection<T>, IEnumerable<T>,
    IList, ICollection, IEnumerable
  {
    ...
    public void Add(T item);
    public ReadOnlyCollection<T> AsReadOnly();
    public int BinarySearch(T item);
    public bool Contains(T item);
    public void CopyTo(T[] array);
    public int FindIndex(System.Predicate<T> match);
    public T FindLast(System.Predicate<T> match);
    public bool Remove(T item);
    public int RemoveAll(System.Predicate<T> match);
    public T[] ToArray();
    public bool TrueForAll(System.Predicate<T> match);
    public T this[int index] { get; set; }
  }
}
```

Когда создается List<T> с указанием объектов Person, это все равно, как если бы тип List<T> был определен следующим образом:

```
namespace System.Collections.Generic
{
  public class List<Person> :
    IList<Person>, ICollection<Person>, IEnumerable<Person>, IReadOnlyList<Person>
    IList, ICollection, IEnumerable
  {
    ...
    public void Add(Person item);
    public ReadOnlyCollection<Person> AsReadOnly();
    public int BinarySearch(Person item);
    public bool Contains(Person item);
    public void CopyTo(Person[] array);
    public int FindIndex(System.Predicate<Person> match);
    public Person FindLast(System.Predicate<Person> match);
    public bool Remove(Person item);
    public int RemoveAll(System.Predicate<Person> match);
    public Person[] ToArray();
    public bool TrueForAll(System.Predicate<Person> match);
    public Person this[int index] { get; set; }
  }
}
```

Разумеется, при создании в коде обобщенной переменной List<T> компилятор на самом деле не создает совершенно новую реализацию класса List<T>. Вместо этого он обрабатывает только члены обобщенного типа, к которым действительно производится обращение.

Указание параметров типа для обобщенных членов

Для необобщенного класса или структуры вполне допустимо поддерживать несколько обобщенных членов (например, методов и свойств). В таких случаях указывать значение заполнителя нужно также и во время вызова метода. Например, System.Array поддерживает несколько обобщенных методов. В частности, статический метод Sort() имеет обобщенный конструктор по имени Sort<T>(). Рассмотрим следующий фрагмент кода, в котором T — это тип int:

```
int[] myInts = { 10, 4, 2, 33, 93 };

// Указание заполнителя для обобщенного метода Sort<>().
Array.Sort<int>(myInts);
foreach (int i in myInts)
{
  Console.WriteLine(i);
}
```

Указание параметров типов для обобщенных интерфейсов

Обобщенные интерфейсы обычно реализуются при построении классов или структур, которые должны поддерживать различные поведения платформы (например, клонирование, сортировку и перечисление). В главе 8 рассматривалось множество необобщенных интерфейсов, таких как IComparable, IEnumerable, IEnumerator и IComparer. Вспомните, как определен необобщенный интерфейс IComparable:

```
public interface IComparable
{
  int CompareTo(object obj);
}
```

В той же главе 8 этот интерфейс был реализован в классе `Car` для обеспечения сортировки в стандартном массиве. Однако код требовал нескольких проверок времени выполнения и операций приведения, потому что параметром был общий тип `System.Object`:

```
public class Car : IComparable
{
  ...
  // Реализация IComparable.
  int IComparable.CompareTo(object obj)
  {
    Car temp = obj as Car;
    if (temp != null)
    {
      if (this.CarID > temp.CarID)
         return 1;
      if (this.CarID < temp.CarID)
         return -1;
      else
         return 0;
    }
    else
      throw new ArgumentException("Parameter is not a Car!");
  }
}
```

Теперь воспользуемся обобщенным аналогом этого интерфейса:

```
public interface IComparable<T>
{
  int CompareTo(T obj);
}
```

В таком случае код реализации будет значительно яснее:

```
public class Car : IComparable<Car>
{
...
  // Реализация IComparable<T>.
  int IComparable<Car>.CompareTo(Car obj)
  {
    if (this.CarID > obj.CarID)
      return 1;
    if (this.CarID < obj.CarID)
      return -1;
    else
      return 0;
  }
}
```

Здесь уже не нужно проверять, относится ли входной параметр к типу `Car`, потому что он может быть *только* `Car`! В случае передачи несовместимого типа данных возникает ошибка на этапе компиляции.

Итак, вы получили начальные сведения о том, как взаимодействовать с обобщенными элементами, а также ознакомились с ролью параметров типа (т.е. заполнителей), и теперь можно приступать к изучению классов и интерфейсов из пространства имен `System.Collections.Generic`.

Пространство имен System.Collections.Generic

Когда выполняется построение приложения .NET и необходим способ управления данным и в памяти, классы из пространства имен `System.Collections.Generic`, скорее всего, удовлетворят всем требованиям. В начале этой главы кратко упоминались некоторые из необобщенных интерфейсов, реализованных необобщенными классами коллекций. Не должно вызывать удивления, что в пространстве имен `System.Collections.Generic` определены обобщенные замены для многих из них.

В действительность есть много обобщенных интерфейсов, которые расширяют свои необобщенные аналоги. Это может показаться странным; однако благодаря этому, реализации новых классов также поддерживают унаследованную функциональность, имеющуюся у их необобщенных аналогов. Например, `IEnumerable<T>` расширяет `IEnumerable`. В табл. 9.4 документированы основные обобщенные интерфейсы, с которыми придется иметь дело при работе с обобщенными классами коллекций.

Таблица 9.4. Основные интерфейсы, поддерживаемые классами из пространства имен System.Collections.Generic

Интерфейс System.Collections.Generic	Назначение
`ICollection<T>`	Определяет общие характеристики (например, размер, перечисление и безопасность к потокам) для всех типов обобщенных коллекций
`IComparer<T>`	Определяет способ сравнения объектов
`IDictionary<TKey, TValue>`	Позволяет объекту обобщенной коллекции представлять свое содержимое посредством пар “ключ/значение”
`IEnumerable<T>`	Возвращает интерфейс `IEnumerator<T>` для заданного объекта
`IEnumerator<T>`	Позволяет выполнять итерацию в стиле `foreach` по элементам коллекции
`IList<T>`	Обеспечивает поведение добавления, удаления и индексации элементов в последовательном списке объектов
`ISet<T>`	Предоставляет базовый интерфейс для абстракции множеств

В пространстве имен `System.Collectiobs.Generic` также определен набор классов, реализующих многие из этих основных интерфейсов. В табл. 9.5 описаны часто используемые классы из этого пространства имен, реализуемые ими интерфейсы и их базовая функциональность.

В пространстве имен `System.Collections.Generic` также определен ряд вспомогательных классов и структур, которые работают в сочетании со специфическим контейнером. Например, тип `LinkedListNode<T>` представляет узел внутри обобщенного контейнера `LinkedList<T>`, исключение `KeyNotFoundException` генерируется при попытке получить элемент из коллекции с указанием несуществующего ключа, и т.д.

Важно отметить, что `mscorlib.dll` и `System.dll` — не единственные сборки, которые добавляют новые типы в пространство имен `System.Collections.Generic`. Например, `System.Core.dll` добавляет класс `HashSet<T>`. Детальные сведения о пространстве имен `System.Collections.Generic` доступны в документации .NET Framework.

Таблица 9.5. Классы из пространства имен `System.Collections.Generic`

Обобщенный класс	Поддерживаемые основные интерфейсы	Назначение
`Dictionary<TKey, TValue>`	`ICollection<T>`, `IDictionary<TKey, TValue>`, `IEnumerable<T>`	Представляет обобщенную коллекцию ключей и значений
`LinkedList<T>`	`ICollection<T>`, `IEnumerable<T>`	Представляет двухсвязный список
`List<T>`	`ICollection<T>`, `IEnumerable<T>`, `IList<T>`	Последовательный список элементов с динамически изменяемым размером
`Queue<T>`	`ICollection` (Это не опечатка! Именно так называется необобщенный интерфейс коллекции), `IEnumerable<T>`	Обобщенная реализация очереди — списка, работающего по алгоритму "первый вошел — первый вышел" (FIFO)
`SortedDictionary<TKey, TValue>`	`ICollection<T>`, `IDictionary<TKey, TValue>`, `IEnumerable<T>`	Обобщенная реализация словаря — отсортированного множества пар "ключ/значение"
`SortedSet<T>`	`ICollection<T>`, `IEnumerable<T>`, `ISet<T>`	Представляет коллекцию объектов, поддерживаемых в сортированном порядке без дублирования
`Stack<T>`	`ICollection` (Это не опечатка! Это интерфейс необобщенной коллекции!), `IEnumerable<T>`	Обобщенная реализация стека — списка, работающего по алгоритму "последний вошел — первый вышел" (LIFO)

В любом случае следующая задача заключается в том, чтобы научиться использовать некоторые из этих обобщенных классов коллекций. Но прежде давайте рассмотрим языковые средства C# (впервые появившиеся в .NET 3.5), которые упрощают наполнение данными обобщенных (и необобщенных) коллекций.

Синтаксис инициализации коллекций

В главе 4 был представлен *синтаксис инициализации объектов*, который позволяет устанавливать свойства для новой переменной во время ее конструирования. С ним тесно связан *синтаксис инициализации коллекций*. Это средство языка C# позволяет наполнять множество контейнеров (таких как `ArrayList` или `List<T>`) элементами с использованием синтаксиса, похожего на тот, что применяется для наполнения базового массива.

На заметку! Синтаксис инициализации коллекций может применяться только к классам, которые поддерживают метод `Add()`, формализованный интерфейсами `ICollection<T>`/`ICollection`.

Рассмотрим следующие примеры:

```
// Инициализация стандартного массива.
int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
// Инициализация обобщенного списка List<> элементов int.
List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Инициализация ArrayList числовыми данными.
ArrayList myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Если контейнер управляет коллекцией классов или структур, можно смешивать синтаксис инициализации объектов с синтаксисом инициализации коллекций, создавая некоторый функциональный код. Возможно, вы помните класс `Point` из главы 5, в котором были определены два свойства `X` и `Y`. Чтобы построить обобщенный список `List<T>` объектов `P`, можно написать такой код:

```
List<Point> myListOfPoints = new List<Point>
{
  new Point { X = 2, Y = 2 },
  new Point { X = 3, Y = 3 },
  new Point(PointColor.BloodRed){ X = 4, Y = 4 }
};

foreach (var pt in myListOfPoints)
{
  Console.WriteLine(pt);
}
```

Преимущество этого синтаксиса в экономии большого объема клавиатурного ввода. Хотя вложенные фигурные скобки затрудняют чтение, если не позаботиться о форматировании, только представьте себе объем кода, который потребовалось бы написать для наполнения следующего списка `List<T>` объектов `Rectangle`, если бы не было синтаксиса инициализации коллекций (вспомните, как в главе 4 создавался класс `Rectangle`, который содержал два свойства, инкапсулирующих объекты `Point`):

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
  new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
                 BottomRight = new Point { X = 200, Y = 200}},
  new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
                 BottomRight = new Point { X = 100, Y = 100}},
  new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
                 BottomRight = new Point { X = 90, Y = 75}}
};

foreach (var r in myListOfRects)
{
  Console.WriteLine(r);
}
```

Работа с классом `List<T>`

Для начала создадим новый проект консольного приложения по имени `FunWithGenericCollections`. Обратите внимание, что в первоначальном файле кода C# пространство имен `System.Collections.Generic` уже импортировано.

Первый обобщенный класс, который мы рассмотрим — это `List<T>`, который уже применялся ранее в этой главе. Из всех классов пространства имен `System.Collections.Generic` класс `List<T>` будет использоваться наиболее часто, потому что он позволяет динамически изменять размер контейнера. Чтобы проиллюстрировать основы этого типа, добавьте в класс `Program` метод `UseGenericList()`, в котором `List<T>` применяется для манипуляций множеством объектов `Person`; вы должны помнить, что в классе `Person` определены три свойства (`Age`, `FirstName` и `LastName`) и специальная реализация метода `ToString()`.

```
static void UseGenericList()
{
  // Создать список объектов Person и заполнить его с помощью
  // синтаксиса инициализации объектов/коллекций.
  List<Person> people = new List<Person>()
  {
    new Person {FirstName= "Homer", LastName="Simpson", Age=47},
    new Person {FirstName= "Marge", LastName="Simpson", Age=45},
    new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
    new Person {FirstName= "Bart", LastName="Simpson", Age=8}
  };
  // Вывести на консоль количество элементов в списке.
  Console.WriteLine("Items in list: {0}", people.Count);
  // Выполнить перечисление по списку.
  foreach (Person p in people)
    Console.WriteLine(p);
  // Вставить новую персону.
  Console.WriteLine("\n->Inserting new person.");
  people.Insert(2, new Person { FirstName = "Maggie", LastName = "Simpson", Age = 2 });
  Console.WriteLine("Items in list: {0}", people.Count);
  // Скопировать данные в новый массив.
  Person[] arrayOfPeople = people.ToArray();
  for (int i = 0; i < arrayOfPeople.Length; i++)
  {
    Console.WriteLine("First Names: {0}", arrayOfPeople[i].FirstName);
  }
}
```

Здесь вы используете синтаксис инициализации для наполнения вашего `List<T>` объектами как сокращенную нотацию вызовов `Add()` *множество* раз. После вывода количества элементов в коллекции (а также перечисления по всем элементам) производится вызов `Insert()`. Как можно видеть, `Insert()` позволяет вставить новый элемент в `List<T>` по указанному индексу.

И, наконец, обратите внимание на вызов метода `ToArray()`, который возвращает массив объектов `Person`, основанный на содержимом исходного `List<T>`. Затем осуществляется проход по всем элементам этого массива с использованием синтаксиса индексатора массива. Если вы вызовете этот метод из `Main()`, то получите следующий вывод:

```
***** Fun with Generic Collections *****

Items in list: 4
Name: Homer Simpson, Age: 47
Name: Marge Simpson, Age: 45
Name: Lisa Simpson, Age: 9
Name: Bart Simpson, Age: 8

->Inserting new person.
Items in list: 5
First Names: Homer
First Names: Marge
First Names: Maggie
First Names: Lisa
First Names: Bart
```

В классе `List<T>` определено множество дополнительных членов, представляющих интерес, поэтому за дополнительной информацией обращайтесь в документацию .NET Framework. Теперь рассмотрим еще несколько обобщенных коллекций: `Stack<T>`, `Queue<T>` и `SortedSet<T>`. Это должно дать более полное понимание базовых вариантов хранения данных в приложении.

Работа с классом Stack<T>

Класс Stack<T> представляет коллекцию элементов, работающую по алгоритму "последний вошел — первый вышел" (LIFO). Как и можно было ожидать, в Stack<T> определены члены Push() и Pop(), предназначенные для вставки и удаления элементов в стеке. Приведенный ниже метод создает стек объектов Person:

```
static void UseGenericStack()
{
  Stack<Person> stackOfPeople = new Stack<Person>();
  stackOfPeople.Push(new Person
    { FirstName = "Homer", LastName = "Simpson", Age = 47 });
  stackOfPeople.Push(new Person
    { FirstName = "Marge", LastName = "Simpson", Age = 45 });
  stackOfPeople.Push(new Person
    { FirstName = "Lisa", LastName = "Simpson", Age = 9 });

  // Просмотреть верхний элемент, вытолкнуть его и просмотреть снова.
  Console.WriteLine("First person is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
  Console.WriteLine("\nFirst person is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
  Console.WriteLine("\nFirst person item is: {0}", stackOfPeople.Peek());
  Console.WriteLine("Popped off {0}", stackOfPeople.Pop());

  try
  {
    Console.WriteLine("\nFirst person is: {0}", stackOfPeople.Peek());
    Console.WriteLine("Popped off {0}", stackOfPeople.Pop());
  }
  catch (InvalidOperationException ex)
  {
    Console.WriteLine("\nError! {0}", ex.Message); // Ошибка! Стек пуст.
  }
}
```

В коде строится стек, содержащий информацию о трех людях, добавленных в порядке их имен: Homer, Marge и Lisa. Заглядывая (посредством Peek()) в стек, вы всегда видите объект, находящийся на его вершине; поэтому первый вызов Peek() вернет третий объект Person. После серии вызовов Pop() и Peek() стек, наконец, опустошается, после чего вызовы Peek() и Pop() приводят к генерации системного исключения. Вывод этого примера показан ниже:

```
***** Fun with Generic Collections *****

First person is: Name: Lisa Simpson, Age: 9
Popped off Name: Lisa Simpson, Age: 9

First person is: Name: Marge Simpson, Age: 45
Popped off Name: Marge Simpson, Age: 45

First person item is: Name: Homer Simpson, Age: 47
Popped off Name: Homer Simpson, Age: 47

Error! Stack empty.
```

Работа с классом Queue<T>

Очереди — это контейнеры, гарантирующие доступ к элементам в стиле "первый вошел — первый вышел" (FIFO). К сожалению, людям приходится сталкиваться с очередями каждый день: очереди в банк, очереди в кинотеатр, очереди в кафе. Когда нужно смоделировать сценарий, в котором элементы обрабатываются в режиме FIFO, класс

Queue<T> подходит наилучшим образом. В дополнение к функциональности, предоставляемой поддерживаемыми интерфейсами, Queue определяет основные члены, которые перечислены в табл. 9.6.

Таблица 9.6. Члены типа Queue<T>

Член Queue<T>	Назначение
Dequeue()	Удаляет и возвращает объект из начала Queue<T>
Enqueue()	Добавляет объект в конец Queue<T>
Peek()	Возвращает объект из начала Queue<T>, не удаляя его

Теперь давайте посмотрим на эти методы в работе. Можно снова вернуться к классу Person и построить объект Queue<T>, эмулирующий очередь людей, которые ожидают заказа кофе. Для начала представим, что имеется следующий статический метод:

```
static void GetCoffee(Person p)
{
  Console.WriteLine("{0} got coffee!", p.FirstName);
}
```

Кроме того, есть также дополнительный вспомогательный метод, который вызывает GetCoffee() внутренне:

```
static void UseGenericQueue()
{
  // Создать очередь из трех человек.
  Queue<Person> peopleQ = new Queue<Person>();
  peopleQ.Enqueue(new Person {FirstName= "Homer",
    LastName="Simpson", Age=47});
  peopleQ.Enqueue(new Person {FirstName= "Marge",
    LastName="Simpson", Age=45});
  peopleQ.Enqueue(new Person {FirstName= "Lisa",
    LastName="Simpson", Age=9});

  // Кто первый в очереди?
  Console.WriteLine("{0} is first in line!", peopleQ.Peek().FirstName);

  // Удалить всех из очереди.
  GetCoffee(peopleQ.Dequeue());
  GetCoffee(peopleQ.Dequeue());
  GetCoffee(peopleQ.Dequeue());

  // Попробовать извлечь кого-то из очереди снова.
  try
  {
    GetCoffee(peopleQ.Dequeue());
  }
  catch(InvalidOperationException e)
  {
    Console.WriteLine("Error! {0}", e.Message); // Ошибка! Очередь пуста.
  }
}
```

Здесь вы вставляете три элемента в класс Queue<T>, используя метод Enqueue(). Вызов Peek() позволяет просматривать (но не удалять) первый элемент, находящийся в данный момент в Queue. Наконец, вызов Dequeue() удаляет элемент из очереди и посылает его вспомогательной функции GetCoffee() для обработки. Обратите внимание, что если вы пытаетесь удалять элементы из пустой очереди, генерируется исключение

времени выполнения. Ниже приведен вывод, который будет получен при вызове этого метода:

```
***** Fun with Generic Collections *****

Homer is first in line!
Homer got coffee!
Marge got coffee!
Lisa got coffee!
Error! Queue empty.
```

Работа с классом SortedSet<T>

Класс SortedSet<T> удобен тем, что при вставке или удалении элементов он автоматически обеспечивает сортировку элементов в наборе. Класс SortedSet<T> понадобится информировать о том, *как* должны сортироваться объекты, за счет передачи его конструктору аргумента — объекта, реализующего обобщенный интерфейс IComparer<T>.

Начнем с создания нового класса по имени SortPeopleByAge, реализующего IComparer<T>, где T — тип Person. Вспомните, что этот интерфейс определяет единственный метод по имени Compare(), в котором можно запрограммировать логику сравнения элементов. Ниже приведена простая реализация этого класса:

```
class SortPeopleByAge : IComparer<Person>
{
  public int Compare(Person firstPerson, Person secondPerson)
  {
    if (firstPerson.Age > secondPerson.Age)
      return 1;
    if (firstPerson.Age < secondPerson.Age)
      return -1;
    else
      return 0;
  }
}
```

Теперь добавим в класс Program следующий новый метод, который должен будет вызван в Main():

```
static void UseSortedSet()
{
  // Создать несколько людей разного возраста.
  SortedSet<Person> setOfPeople = new SortedSet<Person>(new SortPeopleByAge())
  {
    new Person {FirstName= "Homer", LastName="Simpson", Age=47},
    new Person {FirstName= "Marge", LastName="Simpson", Age=45},
    new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
    new Person {FirstName= "Bart", LastName="Simpson", Age=8}
  };

  // Обратите внимание, что элементы отсортированы по возрасту.
  foreach (Person p in setOfPeople)
  {
    Console.WriteLine(p);
  }
  Console.WriteLine();

  // Добавить еще несколько людей разного возраста.
  setOfPeople.Add(new Person { FirstName = "Saku", LastName = "Jones", Age = 1 });
  setOfPeople.Add(new Person { FirstName = "Mikko", LastName = "Jones", Age = 32 });
```

```
    // Элементы по-прежнему отсортированы по возрасту.
    foreach (Person p in setOfPeople)
    {
      Console.WriteLine(p);
    }
  }
```

После запуска приложения видно, что список объектов будет всегда упорядочен по значению свойства `Age`, независимо от порядка вставки и удаления объектов в коллекцию:

```
***** Fun with Generic Collections *****

Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47

Name: Saku Jones, Age: 1
Name: Bart Simpson, Age: 8
Name: Lisa Simpson, Age: 9
Name: Mikko Jones, Age: 32
Name: Marge Simpson, Age: 45
Name: Homer Simpson, Age: 47
```

Исходный код. Проект `FunWithGenericCollections` доступен в подкаталоге `Chapter 09`.

Пространство имен `System.Collections.ObjectModel`

Теперь, когда вы понимаете основы работы с обобщенными классами, мы можем кратко взглянуть на дополнительное пространство имен, связанное с коллекциями — `System.Collections.ObjectModel`. Это относительно небольшое пространство имен, содержащее лишь горстку классов. В табл. 9.7 документированы два класса, которые вы должны знать обязательно.

Таблица 9.7. Полезные типы в `System.Collections.ObjectModel`

Тип `System.Collections.ObjectModel`	Назначение
`ObservableCollection<T>`	Представляет динамическую коллекцию данных, которая обеспечивает уведомления при добавлении элементов, их удалении и обновлении всего списка
`ReadOnlyObservableCollection<T>`	Представляет версию `ObservableCollection<T>`, предназначенную только для чтения

Класс `ObservableCollection<T>` очень удобен в том, что он обладает возможностью информировать внешние объекты, когда его содержимое каким-нибудь образом изменяется (как вы могли догадаться, работа с `ReadOnlyObservableCollection<T>` очень похожа, но имеет природу только для чтения).

Работа с `ObservableCollection<T>`

Создадим новое консольное приложение по имени `FunWithObservableCollection` и импортируем в первоначальный файл кода C# пространство имен `System.Collections.ObjectModel`. Во многих отношениях работа с `ObservableCollection<T>` идентичная

работе с List<T>, учитывая то, что оба класса реализуют одни и те же основные интерфейсы. Уникальность класса ObservableCollection<T> состоит в том, что он поддерживает событие по имени CollectionChanged. Это событие будет инициироваться каждый раз, когда вставляется новый элемент, удаляется (или перемещается) текущий элемент либо модифицируется вся коллекция целиком.

Подобно любому событию, CollectionChanged определено в терминах делегата, которым в данном случае является NotifyCollectionChangedEventHandler. Этот делегат может вызывать любой метод, принимающий объект в первом параметре и NotifyCollectionChangedEventArgs — во втором. Рассмотрим следующий метод Main(), который заполняет наблюдаемую коллекцию, содержащую объекты Person, и привязывается к событию CollectionChanged:

```
class Program
{
  static void Main(string[] args)
  {
    // Сделать коллекцию наблюдаемой и добавить в нее несколько объектов Person.
    ObservableCollection<Person> people = new ObservableCollection<Person>()
    {
      new Person{ FirstName = "Peter", LastName = "Murphy", Age = 52 },
      new Person{ FirstName = "Kevin", LastName = "Key", Age = 48 },
    };

    // Привязаться к событию CollectionChanged.
    people.CollectionChanged += people_CollectionChanged;
  }
  static void people_CollectionChanged(object sender,
    System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
  {
    throw new NotImplementedException();
  }
}
```

Входной параметр NotifyCollectionChangedEventArgs определяет два важных свойства, OldItems и NewItems, предоставляющие список элементов, которые имелись в коллекции перед генерацией события, и новых элементов, которые участвовали в изменении. Тем не менее, эти списки будут исследоваться только при подходящих обстоятельствах. Вспомните, что событие CollectionChanged может инициироваться, когда элементы добавляются, удаляются, перемещаются или сбрасываются. Чтобы выяснить, какое из этих действий запустило событие, можно воспользоваться свойством Action объекта NotifyCollectionChangedEventArgs. Свойство Action может проверяться на предмет равенства с любым из следующих членов перечисления NotifyCollectionChangedAction:

```
public enum NotifyCollectionChangedAction
{
  Add = 0,
  Remove = 1,
  Replace = 2,
  Move = 3,
  Reset = 4,
}
```

Ниже приведена реализация обработчика событий CollectionChanged, который будет обходить старый и новый наборы, когда элемент вставляется или удаляется из рабочей коллекции:

```
static void people_CollectionChanged(object sender,
  System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
{
  // Выяснить действие, которое привело к генерации события.
  Console.WriteLine("Action for this event: {0}", e.Action);

  // Было что-то удалено.
  if (e.Action == System.Collections.Specialized.NotifyCollectionChangedAction.Remove)
  {
    Console.WriteLine("Here are the OLD items:");
    foreach (Person p in e.OldItems)
    {
      Console.WriteLine(p.ToString());
    }
    Console.WriteLine();
  }

  // Было что-то добавлено.
  if (e.Action == System.Collections.Specialized.NotifyCollectionChangedAction.Add)
  {
    // Теперь вывести новые элементы, которые были вставлены.
    Console.WriteLine("Here are the NEW items:");
    foreach (Person p in e.NewItems)
    {
      Console.WriteLine(p.ToString());
    }
  }
}
```

Теперь, предполагая, что вы обновили метод `Main()` для добавления и удаления элемента, вы увидите следующий вывод:

```
Action for this event: Add
Here are the NEW items:
Name:  Fred Smith, Age: 32

Action for this event: Remove
Here are the OLD items:
Name: Peter Murphy, Age: 52
```

На этом исследование различных пространств имен, связанных с коллекциями, в библиотеках базовых классов .NET завершено. В конце этой главы будет также показано, как и для чего строить собственные обобщенные методы и обобщенные типы.

Исходный код. Проект `FunWithObservableCollection` доступен в подкаталоге `Chapter 09`.

Создание специальных обобщенных методов

Хотя большинство разработчиков обычно используют существующие обобщенные типы из библиотек базовых классов, можно также строить собственные обобщенные методы и специальные обобщенные типы. Чтобы понять, как включать обобщения в собственные проекты, начнем с построения обобщенного метода обмена, предварительно создав новое консольное приложение по имени `CustomGenericMethods`.

Построение специальных обобщенных методов представляет собой более развитую версию традиционной перегрузки методов. В главе 2 было показано, что перегрузка — это определение нескольких версий одного метода, отличающихся друг от друга количеством или типами параметров.

Хотя перегрузка — полезное средство объектно-ориентированного языка, при этом возникает проблема, вызванная появлением огромного количества методов, которые в конечном итоге делают одно и то же. Например, предположим, что требуется создать методы, которые позволяют менять местами два фрагмента данных. Можно начать с написания простого метода для обмена двух целочисленных значений:

```
// Обмен двух значений int.
static void Swap(ref int a, ref int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Пока что все хорошо. А теперь представим, что нужно поменять местами два объекта `Person`; для этого понадобится новая версия метода `Swap()`:

```
// Обмен двух объектов Person.
static void Swap(ref Person a, ref Person b)
{
  Person temp;
  temp = a;
  a = b;
  b = temp;
}
```

Уже должно стать ясно, куда это приведет. Если также потребуется поменять местами два значения с плавающей точкой, две битовые карты, два объекта автомобилей или еще что-нибудь, придется писать дополнительные методы, что в конечном итоге превратится в кошмар при сопровождении. Правда, можно было бы построить один (необобщенный) метод, оперирующий параметрами типа `object`, но тогда возникнут проблемы, которые были описаны ранее в этой главе, т.е. упаковка, распаковка, недостаток безопасности типов, явное приведение и т.п.

Всякий раз, когда имеется группа перегруженных методов, отличающихся только входными аргументами — это явный признак того, что за счет применения обобщений удастся облегчить себе жизнь. Рассмотрим следующий обобщенный метод `Swap<T>`, который может менять местами два значения `T`:

```
// Этот метод обменивает между собой значения двух
// элементов типа, переданного в параметре <T>.
static void Swap<T>(ref T a, ref T b)
{
  Console.WriteLine("You sent the Swap() method a {0}",
    typeof(T));
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

Обратите внимание, что обобщенный метод определен за счет спецификации параметра типа после имени метода и перед списком параметров. Здесь устанавливается, что метод `Swap()` может оперировать любыми двумя параметрами типа `<T>`. Чтобы немного прояснить картину, имя подставляемого типа выводится на консоль с использованием операции `typeof()`. Теперь рассмотрим следующий метод `Main()`, обменивающий значениями целочисленные и строковые переменные:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Generic Methods *****\n");

  // Обмен двух значений int.
  int a = 10, b = 90;
  Console.WriteLine("Before swap: {0}, {1}", a, b);
  Swap<int>(ref a, ref b);
  Console.WriteLine("After swap: {0}, {1}", a, b);
  Console.WriteLine();

  // Обмен двух строк.
  string s1 = "Hello", s2 = "There";
  Console.WriteLine("Before swap: {0} {1}!", s1, s2);
  Swap<string>(ref s1, ref s2);
  Console.WriteLine("After swap: {0} {1}!", s1, s2);

  Console.ReadLine();
}
```

Ниже показан вывод этого примера:

```
***** Fun with Custom Generic Methods *****

Before swap: 10, 90
You sent the Swap() method a System.Int32
After swap: 90, 10

Before swap: Hello There!
You sent the Swap() method a System.String
After swap: There Hello!
```

Основное преимущество этого подхода в том, что нужно будет сопровождать только одну версию `Swap<T>()`, хотя она может оперировать любыми двумя элементами определенного типа, причем в безопасной к типам манере. Еще лучше то, что находящиеся в стеке элементы остаются в стеке, а расположенные в куче — соответственно, в куче.

Выведение параметров типа

При вызове таких обобщенных методов, как `Swap<T>`, можно опускать параметр типа, если (и только если) обобщенный метод требует аргументов, поскольку компилятор может вывести параметр типа из параметров членов. Например, добавив к `Main()` следующий код, можно обменивать значения `System.Boolean`:

```
// Компилятор самостоятельно выведет тип System.Boolean.
bool b1 = true, b2 = false;
Console.WriteLine("Before swap: {0}, {1}", b1, b2);
Swap(ref b1, ref b2);
Console.WriteLine("After swap: {0}, {1}", b1, b2);
```

Несмотря на то что компилятор может определить параметр типа на основе типа данных, использованного в объявлении `b1` и `b2`, стоит выработать привычку всегда указывать параметр типа явно:

```
Swap<string>(ref b1, ref b2);
```

Это позволит понять неопытным программистам, что данный метод на самом деле является обобщенным. Более того, выведение типов параметров работает только в том случае, если обобщенный метод принимает, по крайней мере, один параметр.

Например, предположим, что в классе `Program` определен следующий обобщенный метод:

```
static void DisplayBaseClass<T>()
{
  // BaseType — это метод, используемый в рефлексии;
  // он будет рассматриваться в главе 15.
  Console.WriteLine("Base class of {0} is: {1}.",
    typeof(T), typeof(T).BaseType);
}
```

При его вызове потребуется указать параметр типа:

```
static void Main(string[] args)
{
...
  // Необходимо указать параметр типа,
  // если метод не принимает параметров.
  DisplayBaseClass<int>();
  DisplayBaseClass<string>();

  // Ошибка на этапе компиляции! Нет параметров?
  // Значит, необходимо указать тип для подстановки!
  // DisplayBaseClass();
  Console.ReadLine();
}
```

В настоящее время обобщенные методы `Swap<T>` и `DisplayBaseClass<T>` определены в классе `Program` приложения. Конечно, как и любой другой метод, если вы захотите определить эти члены в отдельном классе (`MyGenericMethods`), то можно поступить так:

```
public static class MyGenericMethods
{
  public static void Swap<T>(ref T a, ref T b)
  {
    Console.WriteLine("You sent the Swap() method a {0}",
      typeof(T));
    T temp;
    temp = a;
    a = b;
    b = temp;
  }

  public static void DisplayBaseClass<T>()
  {
    Console.WriteLine("Base class of {0} is: {1}.",
      typeof(T), typeof(T).BaseType);
  }
}
```

Статические методы `Swap<T>` и `DisplayBaseClass<T>` находятся в контексте нового типа статического класса, поэтому потребуется указать имя типа при вызове каждого члена, например:

```
MyGenericMethods.Swap<int>(ref a, ref b);
```

Разумеется, методы не обязательно должны быть статическими. Если бы `Swap<T>` и `DisplayBaseClass<T>` были методами уровня экземпляра (и определенными в нестатическом классе), понадобилось бы просто создать экземпляр `MyGenericMethods` и вызывать их с использованием объектной переменной:

```
MyGenericMethods c = new MyGenericMethods();
c.Swap<int>(ref a, ref b);
```

Исходный код. Проект `CustomGenericMethods` доступен в подкаталоге `Chapter 09`.

Создание специальных обобщенных структур и классов

Теперь, когда известно, как определяются и вызываются обобщенные методы, давайте посмотрим, каким образом сконструировать обобщенную структуру (процесс построения обобщенного класса идентичен) в новом проекте консольного приложения по имени GenericPoint. Предположим, что строится обобщенная структура Point, которая поддерживает единственный параметр типа, определяющий внутреннее представление координат (*x*, *y*). Вызывающий код должен иметь возможность создавать типы Point<T> следующим образом:

```
// Точка с координатами int.
Point<int> p = new Point<int>(10, 10);

// Точка с координатами double.
Point<double> p2 = new Point<double>(5.4, 3.3);
```

Вот полное определение Point<T> с последующим анализом:

```
// Обобщенная структура Point.
public struct Point<T>
{
  // Обобщенные данные состояния.
  private T xPos;
  private T yPos;

  // Обобщенный конструктор.
  public Point(T xVal, T yVal)
  {
    xPos = xVal;
    yPos = yVal;
  }

  // Обобщенные свойства.
  public T X
  {
    get { return xPos; }
    set { xPos = value; }
  }

  public T Y
  {
    get { return yPos; }
    set { yPos = value; }
  }

  public override string ToString()
  {
    return string.Format("[{0}, {1}]", xPos, yPos);
  }

  // Сбросить поля в стандартные значения
  // для заданного параметра типа.
  public void ResetPoint()
  {
    xPos = default(T);
    yPos = default(T);
  }
}
```

Ключевое слово `default` в обобщенном коде

Как видите, структура `Point<T>` использует параметр типа в определении данных полей, аргументов конструктора и определении свойств. Обратите внимание, что в дополнение к переопределению `ToString()`, в `Point<T>` определен метод по имени `ResetPoint()`, в котором применяется не встречавшийся ранее новый синтаксис:

```
// Ключевое слово default в языке C# перегружено.
// При использовании с обобщениями оно представляет
// стандартное значение для параметра типа.
public void ResetPoint()
{
  X = default(T);
  Y = default(T);
}
```

С появлением обобщений ключевое слово `default` обрело второй смысл. В дополнение к использованию с конструкцией `switch`, оно теперь может применяться для установки стандартного значения для параметра типа. Это очень удобно, учитывая, что обобщенный тип не знает заранее, что будет подставлено вместо заполнителя в угловых скобках, и потому не может безопасно строить предположения о стандартных значениях. Умолчания для параметров типа следующие:

1. стандартное значение числовых величин равно `0`;
2. ссылочные типы имеют стандартное значение `null`;
3. поля структур устанавливаются в `0` (для типов значений) или в `null` (для ссылочных типов).

Для `Point<T>` можно было установить значение `X` и `Y` в `0` напрямую, исходя из предположения, что вызывающий код будет применять только числовые значения. Однако за счет использования синтаксиса `default(T)` повышается общая гибкость обобщенного типа. В любом случае теперь можно применять методы `Point<T>` следующим образом:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Generic Structures *****\n");

  // Объект Point, в котором используются int.
  Point<int> p = new Point<int>(10, 10);
  Console.WriteLine("p.ToString()={0}", p.ToString());
  p.ResetPoint();
  Console.WriteLine("p.ToString()={0}", p.ToString());
  Console.WriteLine();
  // Объект Point, в котором используются double.
  Point<double> p2 = new Point<double>(5.4, 3.3);
  Console.WriteLine("p2.ToString()={0}", p2.ToString());
  p2.ResetPoint();
  Console.WriteLine("p2.ToString()={0}", p2.ToString());
  Console.ReadLine();
}
```

Ниже показан вывод этого примера:

```
***** Fun with Generic Structures *****

p.ToString()=[10, 10]
p.ToString()=[0, 0]

p2.ToString()=[5.4, 3.3]
p2.ToString()=[0, 0]
```

Исходный код. Проект `GenericPoint` доступен в подкаталоге `Chapter 09`.

Ограничение параметров типа

Как показано в этой главе, любой обобщенный элемент имеет, по крайней мере, один параметр типа, который должен быть указан при взаимодействии с обобщенным типом или членом. Одно это позволит строить безопасный в отношении типов код; однако платформа .NET позволяет использовать ключевое слово `where` для указания особых требований к определенному параметру типа.

С помощью ключевого слова `this` можно добавлять набор ограничений к конкретному параметру типа, которые компилятор C# проверит во время компиляции. В частности, параметр типа можно ограничить, как описано в табл. 9.8.

Таблица 9.8. Возможные ограничения параметров типа для обобщений

Ограничение обобщения	Назначение
`where T : struct`	Параметр типа <T> должен иметь в своей цепочке наследования `System.ValueType` (т.е. <T> должен быть структурой)
`where T : class`	Параметр типа <T> *не* должен иметь `System.ValueType` в своей цепочке наследования (т.е. <T> должен быть ссылочным типом)
`where T : new()`	Параметр типа <T> должен иметь стандартный конструктор. Это полезно, если обобщенный тип должен создавать экземпляры параметра типа, поскольку не удается предположить формат специальных конструкторов. Обратите внимание, что в типе с несколькими ограничениями это ограничение должно указываться последним
`where T :` *ИмяБазовогоКласса*	Параметр типа <T> должен быть наследником класса, указанного в *ИмяБазовогоКласса*
`where T :` *ИмяИнтерфейса*	Параметр типа <T> должен реализовать интерфейс, указанный в *ИмяИнтерфейса*. Можно задавать несколько интерфейсов, разделяя их запятыми

Если только не требуется строить какие-то исключительно безопасные к типам специальные коллекции, возможно, никогда не придется использовать ключевое слово `where` в проектах C#. Так или иначе, но в следующих нескольких примерах (частичного) кода демонстрируется работа с ключевым словом `where`.

Примеры использования ключевого слова `where`

Будем исходить из того, что создан специальный обобщенный класс, и необходимо гарантировать наличие в параметре типа стандартного конструктора. Это может быть полезно, когда специальный обобщенный класс должен создавать экземпляры `T`, потому что стандартный конструктор — это единственный конструктор, потенциально общий для всех типов. Также подобного рода ограничение `T` позволит производить проверку во время компиляции; если `T` — ссылочный тип, то компилятор напомнит программисту о необходимости переопределения стандартного конструктора в объявлении класса (если помните, стандартные конструкторы удаляются из классов, в которых определены собственные конструкторы).

```
// Класс MyGenericClass унаследован от object, причем содержащиеся
// в нем элементы должны иметь стандартный конструктор.
public class MyGenericClass<T> where T : new()
{
...
}
```

Обратите внимание, что конструкция `where` указывает параметр типа, на который накладывается ограничение, а за ним следует операция двоеточия. После этой операции перечисляются все возможные ограничения (в данном случае — стандартный конструктор). Ниже показан еще один пример:

```
// MyGenericClass унаследован от Object, причем содержащиеся
// в нем элементы должны относиться к классу, реализующему IDrawable,
// и поддерживать стандартный конструктор.
public class MyGenericClass<T> where T : class, IDrawable, new()
{...}
```

В данном случае к `T` предъявляются три требования. Во-первых, это должен быть ссылочный тип (не структура), что помечено лексемой `class`. Во-вторых, `T` должен реализовывать интерфейс `IDrawable`. В-третьих, он также должен иметь стандартный конструктор. Множество ограничений перечисляются в списке, разделенном запятыми; однако имейте в виду, что ограничение `new()` всегда должно идти последним! По этой причине следующий код не скомпилируется:

```
// Ошибка! Ограничение new() должно быть последним в списке!
public class MyGenericClass<T> where T : new(), class, IDrawable
{
  ...
}
```

В случае создания обобщенного класса коллекции с несколькими параметрами типа можно указывать уникальный набор ограничений для каждого параметра с помощью отдельной конструкции `where`:

```
// <K> должен расширять SomeBaseClass и иметь стандартный конструктор, в то время
// как <T> должен быть структурой и реализовывать обобщенный интерфейс IComparable.
public class MyGenericClass<K, T> where K : SomeBaseClass, new()
  where T : IComparable<T>
{
  ...
}
```

Необходимость построения полностью нового обобщенного класса коллекции возникает редко; однако ключевое слово `where` также допускается применять и в обобщенных методах. Например, если необходимо гарантировать, чтобы метод `Swap<T>()` работал только со структурами, измените код следующим образом:

```
// Этот метод обменяет местами любые структуры, но не классы.
static void Swap<T>(ref T a, ref T b) where T : struct
{
  ...
}
```

Обратите внимание, что если ограничить метод `Swap()` подобным образом, обменивать местами объекты `string` (как это делалось в коде примера) уже не получится, поскольку `string` является ссылочным типом.

Недостаток ограничений операций

В конце этой главы следует упомянуть об одном моменте относительно обобщенных методов и ограничений. При создании обобщенных методов может оказаться сюрпризом появление ошибок компиляции во время применения любых операций C# (+, –, *, == и т.д.) к параметрам типа. Например, подумайте, насколько полезным был бы класс, который может выполнять операции `Add()`, `Substract()`, `Multiply()` и `Divide()` над обобщенными типами:

```
// Ошибка на этапе компиляции! Нельзя
// применять операции к параметрам типа!
public class BasicMath<T>
{
  public T Add(T arg1, T arg2)
  { return arg1 + arg2; }

  public T Subtract(T arg1, T arg2)
  { return arg1 - arg2; }

  public T Multiply(T arg1, T arg2)
  { return arg1 * arg2; }

  public T Divide(T arg1, T arg2)
  { return arg1 / arg2; }
}
```

К сожалению, приведенный выше класс `BasicMath<T>` не скомпилируется. Хотя это может показаться серьезным недостатком, следует снова вспомнить, что обобщения являются общими. Естественно, числовые данные работают достаточно хорошо с бинарными операциями C#. С другой стороны, если аргумент `<T>` является специальным классом или структурой, то компилятор мог бы предположить, что этот класс или структура поддерживает операции +, –, * и /. В идеале язык C# должен был бы позволять ограничивать обобщенные типы поддерживаемыми операциями, например:

```
// Код только для иллюстрации!
public class BasicMath<T> where T : operator +, operator -,
  operator *, operator /
{
  public T Add(T arg1, T arg2)
  { return arg1 + arg2; }

  public T Subtract(T arg1, T arg2)
  { return arg1 - arg2; }

  public T Multiply(T arg1, T arg2)
  { return arg1 * arg2; }

  public T Divide(T arg1, T arg2)
  { return arg1 / arg2; }
}
```

К сожалению, ограничения операций в текущей версии C# не поддерживаются. Тем не менее, достичь желаемого эффекта можно (хотя это и потребует дополнительной работы) за счет определения интерфейса, который поддерживает эти операции (интерфейсы C# могут определять операции!), и последующего указания ограничения интерфейса для обобщенного класса. На этом первоначальный обзор построения специальных обобщенных типов завершен. В главе 10 мы вновь обратимся к теме обобщений, когда будем рассматривать тип делегата .NET.

Резюме

Эта глава начиналась с исследования необобщенных типов коллекций в пространствах имен `System.Collections` и `System.Collections.Specialized`, включая различные проблемы, которые связаны со многими необобщенными контейнерами, в том числе недостаток безопасности к типам и накладные расходы времени выполнения в форме операций упаковки и распаковки. Как упоминалось в главе, именно по этим причинам в современных приложениях .NET будут использоваться обобщенные классы коллекций из пространств имен `System.Collections.Generic` и `System.Collections.ObjectModel`.

Вы видели, что обобщенный элемент позволяет указывать заполнители (параметры типа), которые задаются во время создания (или вызова — в случае обобщенных методов). Хотя чаще всего будут просто применяться обобщенные типы, предоставляемые библиотеками базовых классов .NET, можно также создавать собственные обобщенные типы (и обобщенные методы). При этом имеется возможность указания любого количества ограничений (с помощью ключевого слова `where`) для повышения уровня безопасности к типам и обеспечения гарантии выполнения операций над типами в *известном объеме*, что позволяет предоставить определенные базовые возможности.

В качестве финального замечания: не забывайте, что обобщения можно обнаружить во многих местах библиотек базовых классов .NET. В настоящей главе мы сосредоточили внимание конкретно на обобщенных коллекциях. Тем не менее, по мере изучения остальных материалов книги (и погружении в платформу с учетом своей специфики), вы наверняка найдете обобщенные классы, структуры и делегаты, расположенные в том или ином пространстве имен. Кроме того, будьте настороже относительно обобщенных членов необобщенного класса!